嵌入式技术与应用丛书

STM32F0 系列 Cortex-M0 原理与实践

张燕妮　主　编

丁维才　副主编

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书以实际应用开发所需要的知识为主线并重点介绍如何解决开发过程中遇到的问题。全书共分为 17 章。首先分析了 STM32F0x 的性能及价格优势所在，即为何选择 STM32F0x，然后书中结合大量实例详细讲解了系统定时器、GPIO、NVIC、UART、I^2C、SPI、ADC、DAC、PWM、定时器、CAN 等外设，使用 STM32F0x 的固件库写各外设的例程，接着讲解 RTX 实时操作系统以及嵌入式程序结构的 4 种模式优缺点对比，最后分析了 USB 电流/电压监测需求，以及如何根据需求设计相应的解决方案。

本书可作为单片机用户的自学用书、嵌入式工程技术人员的学习和培训用书，也可作为大学生学习单片机的教材。

图书在版编目（CIP）数据

STM32F0 系列 Cortex-M0 原理与实践 / 张燕妮主编. —北京：电子工业出版社，2016.2
（嵌入式技术与应用丛书）
ISBN 978-7-121-28086-3

Ⅰ. ①S… Ⅱ. ①张… Ⅲ. ①微处理器 Ⅳ. ①TP332.3

中国版本图书馆 CIP 数据核字（2016）第 012111 号

责任编辑：刘海艳
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：17　字数：435 千字
版　　次：2016 年 2 月第 1 版
印　　次：2016 年 2 月第 1 次印刷
印　　数：3 000 册　　定价：48.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

如果问一个设计人员，产品设计时，选择器件最关心的因素是什么，那么跟得上形势发展可能是常见的、很重要的一个因素。如果问老板选择器件最关心的因素是什么，可能只有“便宜”两个字。ST 公司的 STM32F0x 芯片是一款基于 Cortex-M0，兼顾性能与价格优势于一体的 32 位处理器。ST 公司的 Cortex-M3 系列产品已经在国内拥有大量客户群体，STM32F0x 芯片的用户群也在日益扩大中。

Cortex-M0 芯片具有小型、低功耗、低闸数、精简程序代码的特点，内建各种模拟与混合信号组件及多种高速通信能力器件，开发人员可以直接跳过 16 位系统，以接近 8 位系统的成本开销获取 32 位系统的性能。Cortex-M0 芯片是学习 ARM 处理器的最佳入门选择。

STM32F0x 在 Cortex-M0 芯片基础上继承了 ST 公司 Cortex-M3 系列产品优点，并对一些不足之处进行了修正（例如 I^2C 外设），还增加了一些优秀功能（例如 USART 的超时检测），适用于工业控制器、家庭自动化、打印机和白色家电、游戏机、DVD/蓝光播放机和音频/视频接收机等。

STM32F0x 具有如下特点。

- 百货迎百客，按需选择：STM32F0x 根据外设情况，将产品划分成多种系列，并不是一种大而全的芯片，用户可根据产品需求与成本选用不同芯片。例如，对于是否需要 CAN 通信就限制选择范围与成本。
- 一招鲜吃遍天下：STM32F0x 家族成员比较多，但不代表各自为政。STM32F0x 家族的外设与引脚位置不是一对一。在相同封装下 STM32F0x 芯片引脚定义相同，通过外设复用选用不同外设。而且借助 ST 公司提供 STM32F0x 固件库开发的应用程序可在 STM32F0x 家族（只要包含该外设）随意运行。书中的代码除了 CAN 通信一章，程序可以在各种 STM32F0x 上运行。

全书共 17 章。其中第 1～4 章是基础；第 5～15 章是外设功能讲解；第 16、17 章是综合实例与高级功能。

第 1 章首先讲解相比 8 位机，Cortex-M0 的优势，以及如何从 8 位机过渡到 Cortex-M0，并说明 Cortex-M0 的基础与特征，主要有寄存器、存储器映射、系统总线、存储器保护单元、嵌套中断控制器；最后一部分会说明 STM32F0x 在同类产品的优势以及特点，为何选用 STM32F0x 进行产品的开发。

第 2 章主要讲解了进行 STM32F0 软件开发需要准备的条件，分别介绍了 MDK 以及 J-ink、U-ink、Stlink 仿真器。MDK 是原 Keil 公司的产品，对于熟悉 Keil C51 单片机开发的用户会感受到 ARM 公司对 C8051 用户的重视，也非常欢迎该用户群体过渡到 Cortex-M0 的开发中。

第 3 章讲解有关硬件设计的基础，主要包括 STM32F0x 的电源、时钟，以及本书所使用的电路图。

第 4 章讲解进行 STM32F0x 软件开发的固件库内容。固件库内容涉及了 ARM 公司的 CMSIS 标准及 ST 公司的固件库标准。其中 CMSIS 是理解目前 ARM 对软件组织结构以及系统启动文件的规范，也是目前所有 Cortex-M 内核 CPU 软件设计的要求与基础。最后讲解了使用 ST 公司固件库建立工程文件的过程。

第 5 章～第 14 章讲解了 STM32F0x 的系统定时器、GPIO、NVIC、UART、I^2C、SPI、ADC、DAC、PWM、定时器、CAN。每 1 章均提供了 1～2 个实际项目中的使用实例，涉及实际开发中使用各种芯片的注意事项以及技巧（或者说作者的经验教训）。

第 16 章讲解 RTX 操作系统，介绍 Keil 自带的实时操作系统的原理与使用过程。结合第 4 章的超级循环和中断模式，读者可根据自己产品特点选用相应的软件程序框架。

第 17 章是关于如何使用 STM32F0x 设计一个 USB 的电流、电压检测器，从设计需求、硬件设计、软件设计等方面进行详细分析，引导读者如何做产品设计、开发。

本书的第 1～8 章、第 10～11 章和第 13～17 章由张燕妮编写，第 9 章由谢玲和王洪玲编写，第 12 章由贾芳和丁维才编写，同时丁维才对全书代码进行了验证。

书中讲解的源代码只摘取了与相应章节配套的部分进行了说明，需要完整代码，可从电子工业出版社电子信息出版分社（http://xxdz.phei.com.cn）下载。

感谢电子工业出版社的王敬栋、刘海艳两位编辑对本书的大力支持。感谢家人与朋友的理解和支持。

由于 STM32F0x 家族的新成员仍在不断增加中，并且作者水平有限以及时间仓促，难免有差错和不足之处恳请读者批评指正。

张燕妮

目　录

第 1 章

低成本单片机世界的入侵者——Cortex-M0

Cortex-M0 处理器是为迎合现在超低功耗和混合信号设备的需求而设计的。连接方式的多样化（例如以太网、USB、低功耗的无线）以及模拟传感器的使用（例如触摸传感器和加速度计）都带来了对处理器的新需求，通常这些应用均需高集成的模拟和数字功能来处理传输数据，而现有的 8 位和 16 位机需要付出增大代码量和提高时钟频率的代价才能满足应用要求。

Cortex-M0 可以在保持低功耗、低成本的前提下满足这些需求，因此 Cortex-M0 处理器的用户群体在快速增长。

Cortex-M0 处理器在 2009 年由 ARM 公司推出，向上兼容 ARM 的 Cortex-M3，并且损耗很低。Cortex-M0 在最小配置时仅有 12 000 逻辑门，同 8 位或者 16 位机一样的少，但却是完全的 32 位处理器。

本章将主要介绍 Cortex-M0 的基本原理，主要包含编程模型、存储器模型、异常模型、电源管理、指令集；并重点分析 Cortex-M0 相比 8 位机优势在哪里，以及如何从 8 位机过渡到 Cortex-M0；最后给出了 STM32F0 的特点。

1.1 相比 8 位（16 位）机为何要选择 Cortex-M0

Cortex-M0 处理器基于一个高集成、低功耗的 32 位处理器内核，采用 3 级流水线冯·诺伊曼结构（Von Neumann architecture）。通过简单、功能强大的指令集以及全面优化的设计（提供包括一个单周期乘法器在内的高端处理硬件），Cortex-M0 处理器可实现极高的能效。Cortex-M0 处理器采用 ARMv6-M 结构，使用 16 位的 Thumb 指令集，并包含 Thumb-2 技术，代码密度比 8 位和 16 位机都要高，这意味对于同样的程序可选用较少的 Flash 处理器，从而可节约成本和功耗。Cortex-M0 的执行周期是 0.896DMIPS/MHz，可使用较少的指令周期执行一个任务（即使 32 位乘法也可在一个周期内完成）。即使在处理不同优先级的嵌套中断情况，嵌套中断处理控制器（NVIC）也会使中断花销很少。

图 1-1 是 Cortex-M0 的具体实现框图。

Cortex-M0 处理器紧密集成了一个可配置的嵌套向量中断处理器（NVIC），提供业界领先的中断性能。NVIC 具有以下功能：

- 包含一个不可屏蔽的中断（NMI）；

- 提供零抖动中断选项；
- 提供 4 个中断优先级。

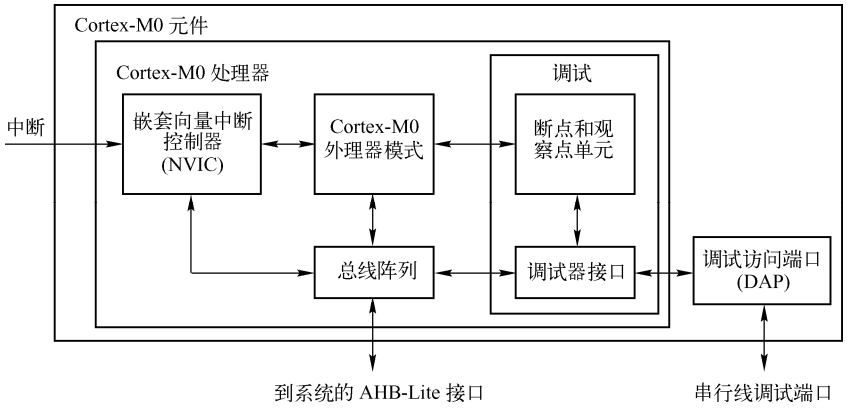


图 1-1　Cortex-M0 的具体实现框图

处理器内核和 NVIC 的紧密结合使得中断服务程序（ISR）快速执行，通过寄存器的硬件堆栈以及加载-乘和存储-乘操作的停止和重启等方式缩短了中断延迟时间。中断处理程序不需要任何汇编封装代码，不用消耗任何 ISR 代码。末尾连锁的优化大大地降低了从一个 ISR 切换到另一个 ISR 时的开销。为了优化低功耗设计，NVIC 还与睡眠模式相结合，提供一个深度睡眠功能，使整个器件能够迅速掉电。

Cortex-M0 处理器提供一个简单的系统级接口，使用 AMBA 技术来提供高速、低延迟的存储器访问。

Cortex-M0 处理器执行一个完整的硬件调试方案，带有大量的硬件断点和观察点选项。通过一个非常适合微控制器和其他小型封装器件的 2 脚串行线调试（SWD）端口，Cortex-M0 提供了一个高度透明的处理器、存储器和外设执行方式。

Cortex-M0 被大量的现代编译器支持。中断服务程序可直接使用 C 语言编码而无须汇编语言。而且指令集仅仅有 56 条指令，便于学习。尽管 Cortex-M0 是一个高性能 pipelined 处理器，指令和中断的执行时序是完全确定的（零抖动），从而使得设计人员可以预测和分析精确时序。支持多种调试方式，除了 ARM 自己公司的 MDK 软件，另有大量第三方软件支持，以及基于 ARM 构建的大量操作系统支持。

1.1.1　性能对比

8 位机或者 16 位机的用户为何要转向 Cortex-M0 呢？Cortex-M0 处理器与典型 16 位机差不多大小，但它具有比 16 位机更好的性能。表 1-1 是基于 DMIPS 的测试对比情况。

表 1-1　几种构架的 DMIPS 对比

体 系 结 构	基于Dhrystone 2.1的预估DMIPS/MHz
最初的80C51	0.0094
PIC18	0.01966

续表

体 系 结 构	基于Dhrystone 2.1的预估DMIPS/MHz
快速8051	0.113
H8S/300H	0.16
HCS12	0.19
MSP430	0.288
H8S/2600	0.303
S12X	0.34
PIC24	0.445
Cortex-M0	0.896

从表 1-1 可以看出，Cortex-M0 的处理速度比所有流行的 16 位处理都快，是最快的 8051 的 8 倍。

注：DMIPS:Dhrystone Million Instructions executed Per Second 主要用于测整数计算能力。Dhrystone: 1976 年发布，主要用于测整数计算能力，Dhrystone 的主体部分是一个由固定顺序指令构成的循环体。Dhrystone 的结果表现形式是微处理器执行固定次数的该循环体所用的时间，用 DMIPS 来表示。DMIPS 标准的优点在于可以比较衡量不同处理器的相对性能，并且被广泛应用于不同处理器间的性能比较。

1.1.2　8 位和 16 位体系结构的缺点

相比 8 位或 16 位机，Cortex-M0 无体系结构的缺陷。下面是 8 位或 16 位机的一些缺陷。

8 位和 16 位机结构的一个明显缺点是内存太小。程序和数据 RAM 太小限制了嵌入式产品的能力。其他的类似堆栈内存太小（例如 8051 堆栈位于内部 RAM，被限制成 256 字节，包含 register bank space）也影响软件设计。ARM 的体系结构的处理器，其内存空间是非常大的，堆栈位于系统内存中，软件设计非常灵活。

许多 8 位和 16 位微控器通过将内存空间划分成内存页方式访问较大内存。软件开发反而因此变得困难，因为在不同的内存页中存取地址不直截了当。因为在不同内存页过渡切换，从而增大了代码尺寸，还降低了性能。然而，ARM 微控器使用 32 位线性地址并不需要内存分页从而更容易使用，并提供更好的开发效率。

8 位微控器体系结构的另外一个缺点是指令集。例如，8051 过度依赖累加器寄存器来处理数据和内存传递，这样增加了代码尺长度。

地址模式也是影响 8 位机和 16 位机性能的一个因素。在 Cortex-M0 中有多种地址模式可用，从而保证代码长度和方便性。

1.1.3　Cortex 的软件移植性

相比 8 位机或者 16 位机，ARM 的 Cortex 处理器的编程是非常方便的。ARM 的 Cortex 微控器可全部使用 C 语言进行软件编程，从而缩短软件开发时间并提高软件的可移植性。即使软件人员计划使用汇编语言，指令集也是非常容易理解的。而且，因为编程模型类似 ARM7TDMI，熟悉 ARM 处理器的人员会很快熟悉 Cortex 微控器。

Cortex-M0 的构架可高效地实现嵌入式操作系统。在复杂应用中，使用嵌入式操作系统可轻松地处理并行任务。并且由于 ARM 是 IP（intellectual property）的供应商，ARM 处理器被广大微控器厂商采用，方便用户选型。

除了硬件，还有大量的嵌入式操作系统、代码库、开发工具以及其他资源供选择。良好的软件生态体系可将精力集中到开发中，提高产品开发速度。

1.2 如何从 8 位机过渡到 Cortex-M0

ARM 公司的 Joseph Yiu 和 Andrew Frame 从堆栈内存、数据类型、非对齐数据等方面分析 8 位机与 Cortex-M 的差异，以及如何从 8 位机过渡到 Cortex-M（包含 Cortex-M0、Cortex-M3、Cortex-M4）。

1．内存空间

ARM 处理器采用 32 位寻址，可实现一个 4GB 的线性内存空间。该内存空间在结构上分成多个区。每个区都有各自的推荐用法（虽然并不是固定的）。统一内存架构不仅增加了内存使用的灵活性，而且降低了不同内存空间使用不同数据类型的复杂性。

相反地，8051 微控制器具有多个内存空间。内存空间的分割增加有效地利用全部内存空间的困难，而且需要借助 C 语言扩展来处理不同的内存类型。

8051 在外部 RAM 内存空间上最高支持 64KB 的程序内存和 64KB 的数据内存。理论上，可以利用内存分页来扩展程序内存大小。不过，内存分页解决方案并未标准化，换句话说，不同 8051 供应商的内存分页的实现并不相同。这不仅会增加软件开发的复杂性，而且由于处理页面切换增加软件开销，显著降低软件性能。

在 ARM Cortex-M3 或 Cortex-M4 上，SRAM 区和外设区都提供了一个 1MB 的位段区（bit band region）。此位段区允许通过别名地址访问其内部的每个位。由于位段别名地址只需通过普通的内存存取指令即可访问，因此被 C 语言完全支持，不需要任何特殊指令。而 8051 提供了少量的位寻址内存（内部 RAM 上 16 字节和 SFR 空间上 16 字节）。处理这些位数据需要特殊指令支持此功能，并且 C 编译器需要扩展。

图 1-2 是 Cortex-M 内存映射方式。图 1-3 是 8051 的内存映射方式。

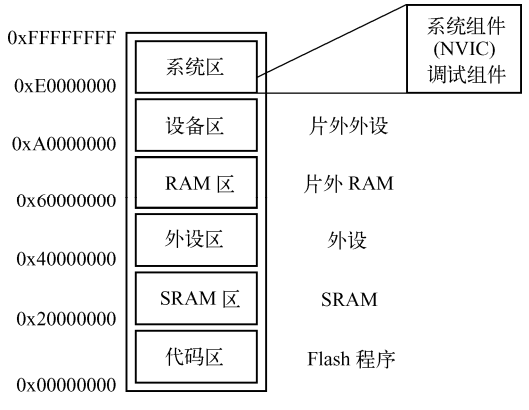


图 1-2　Cortex-M 的内存映射方式

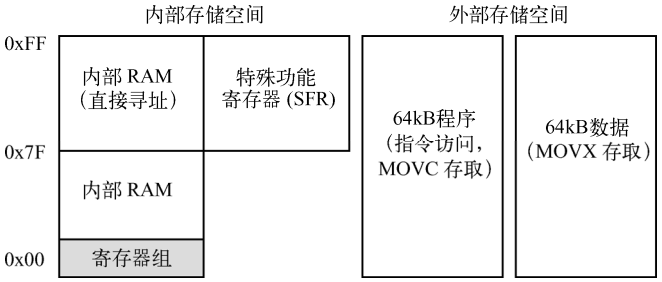


图 1-3　8051 的内存映射方式

2．堆栈内存

堆栈内存操作是内存架构的重要组成部分。在 8051 中，堆栈指针只有 8 位，同时堆栈位于内部的内存空间（上限为 256 个字节，并由工作寄存器和内部数据变量共享）。堆栈操作基于满递增模式。图 1-4 是 8051 的堆栈映射方式。

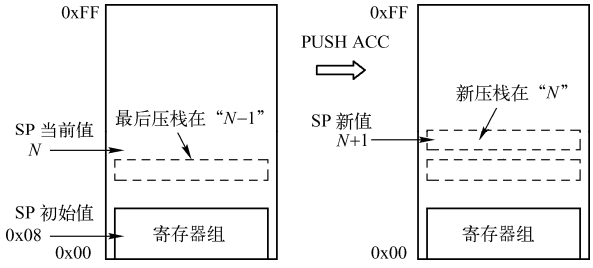


图 1-4　8051 堆栈映射方式

与 8051 不同的是，ARM Cortex-M 处理器使用系统内存作为堆栈，采用满递减模式。图 1-5 是 Cortex-M 堆栈示意图。

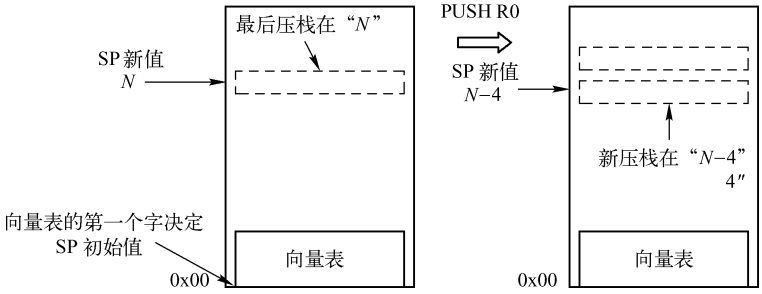


图 1-5　Cortex-M 堆栈示意图

满递减堆栈内存模型更方便 C 语言。例如，微控制器中的 SRAM 的使用可以是使用动态分配内存空间的 C 库。图 1-6 是 Cortex-M 的堆与栈。尽管 Cortex-M 处理器的每次压栈需要 32 位的堆栈内存，RAM 使用总体上仍然要比 8051 小。8051 的变量通常是静态地放在 IDATA 上，而 ARM 处理的局部变量是放在堆栈内存上的，因此，只有当函数执行的时候，局部变量才会占用 RAM 空间。此外，ARM Cortex-M 处理器提供第二个堆栈指针，以允许操作系

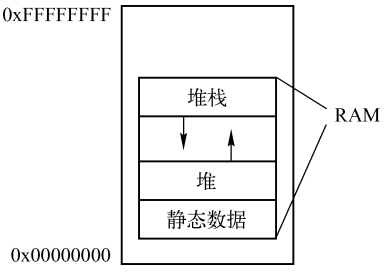


图 1-6　Cortex-M 的堆与栈

统内核和进程堆栈使用不同的堆栈内存。这使得操作更可靠，也使操作系统设计更高效。（堆栈指针切换是自动处理的）

3．数据类型

8051 和 ARM 处理器的数据类型有一些差异。由于数据大小不同，如果程序代码存在依赖数据类型大小或溢出行为，不做修改可能无法工作。表 1-2 为常见数据类型的大小，具体视编译器而定。

表 1-2　8051 与 ARM 中的数据类型

类　　型	8051 中的位数	ARM 中的位数
char,unsigned char	8	8
enum	16	8/16/32（选用最少的一个）
short,unsigned short	16	16
int,unsigned int	16	32
long,unsigned long	32	32

数据类型大小不同的另一个影响是在 ROM 中保留常数数据所需的大小。例如，如果 8051 程序中包含一个整数型常数数组，那么需要修改代码，将该数组定义为短整型常数。否则，代码长度可能会因为该数组从 16 位变成 32 位而增加。由于 8051 架构的特性，8051 的 C 编译器还支持一些数据类型和内存类型扩展，见表 1-3。这些数据类型在 ARM 处理器上是不被支持的。

表 1-3　8051 专有数据类型

类　　型	描　　述	位　　数
bit	在0x20与0x3C之间的位可寻址	1
sbit	在SFR区域的位可寻址	1
sfr	专用功能寄存器	8
sfr16	16位专用功能寄存器	16
idata	内部寻址区的数据	
xdata	外部寻址区数据	
bdata	片内可位寻址数据	

由于 8051 的处理能力限制，大多数 8051 C 编译器会将“双精度”数据类型（64 位）作为单精度（32 位）来处理。而在 ARM 处理器中使用相同代码时，C 编译器将使用双精度，因此程序行为可能会发生变化。例如，如果只需要单精度，需要采用如下类似修改方式进行修改。

```
X=T*atan(T2*sin(X)*cos(X)/(cos(X+Y)+cos(X-Y)-1.0)); /* double precision on ARM*/
Y=T*atan(T2*sin(Y)*cos(Y)/(cos(X+Y)+cos(X-Y)-1.0));
```

对于单精度运算，代码需要更改为：

```
X=T*atanf(T2*sinf(X)*cosf(X)/(cosf(X+Y)+cosf(X-Y)-1.0F));   /*single precision on ARM*/
Y=T*atanf(T2*sinf(Y)*cosf(Y)/(cosf(X+Y)+cosf(X-Y)-1.0F));
```

对于不需要双精度精确度的应用程序，将代码更改为单精度能够提高性能并缩短代码长度。

4．混用 C 语言和汇编程序

大多数情况下，完全可用 C 语言来编写 Cortex-M 应用程序。即使需要访问一些 C 编译器无法通过普通 C 代码生成的特殊指令，也可以使用 CMSIS 提供的隐含函数，或者根据需要在应用程序中使用汇编语言。可以在单独的汇编程序文件中编写汇编代码，也可以使用 C 编译器的特定方法将汇编代码混合在 C 程序文件中。

使用 ARM（和 Keil）开发工具时，将汇编代码插入 C 编程文件的方法称为“嵌入式汇编程序”。汇编代码被声明为函数，并可以被 C 代码调用。例如：

```
int main(void)
{
  int status;
  status =get_primask();
  while(1);
}
__asm int get_primask(void)
{
  MRS R0,PRIMASK;Put interrupt masking register in R0;
  BX LR ;return
}
```

使用 GCC 和 IAR 编译器时，可以使用内嵌汇编程序将汇编代码插入 C 程序代码中。请注意，虽然包括 RVDS 和 KEIL MDK-ARM 在内的 ARM 开发工具中也包含内嵌汇编程序功能，但是 ARM 工具中的内嵌汇编程序仅支持 ARM 指令，并不支持 Thumb 指令，因此不能用于 Cortex-M0 处理器。在汇编程序和 C 语言混合环境中，可以通过汇编程序代码调用 C 函数，也可以通过 C 函数调用汇编程序代码。在简单的情况下，可以使用 R0～R3 作为函数的输入（R0 作为第一个输入变量，以此类推），并使用 R0 来返回结果。函数应该保留 R4～R11 的值，而如果调用 C 函数，那么返回时该 C 函数可能会更改 R0～R3 和 R12 的值。

1.3　编 程 模 型

本节描述 Cortex-M0 的编程模型，主要是关于处理器的模式、堆栈和寄存器的。

1.3.1　处理器的模式

处理器的模式有线程模式（Thread）和处理器模式（handler）两种。其中，线程模式用来执行应用软件，处理器在退出复位时进入线程模式。处理器模式用来处理异常，处理器在完成所有的异常处理后返回到线程模式。Cortex-M0 不支持多特权级，能一直使用所有指令和访问所有资源。

1.3.2 堆栈

处理器使用一个满递减堆栈。这就意味着堆栈指针指向堆栈存储器中的最后一个堆栈项。当处理器将一个新的项压入堆栈时，堆栈指针递减，然后将该项写入新的存储器单元。

处理器执行两个堆栈，主堆栈和进程堆栈，两个堆栈有自己独立的堆栈指针副本。在线程模式下，CONTROL 寄存器控制着处理器使用主堆栈还是进程堆栈。在处理器模式下，处理器总是使用主堆栈。处理器操作的选择见表 1-4。

表 1-4 处理器模式和堆栈使用的选择

处理器的模式	应用情况	使用的堆栈
线程模式	执行应用程序	主堆栈或者进程堆栈。由 CONTROL 寄存器控制
处理器模式	执行异常处理程序	主堆栈

1.3.3 内核寄存器

图 1-7 是处理器内核寄存器。

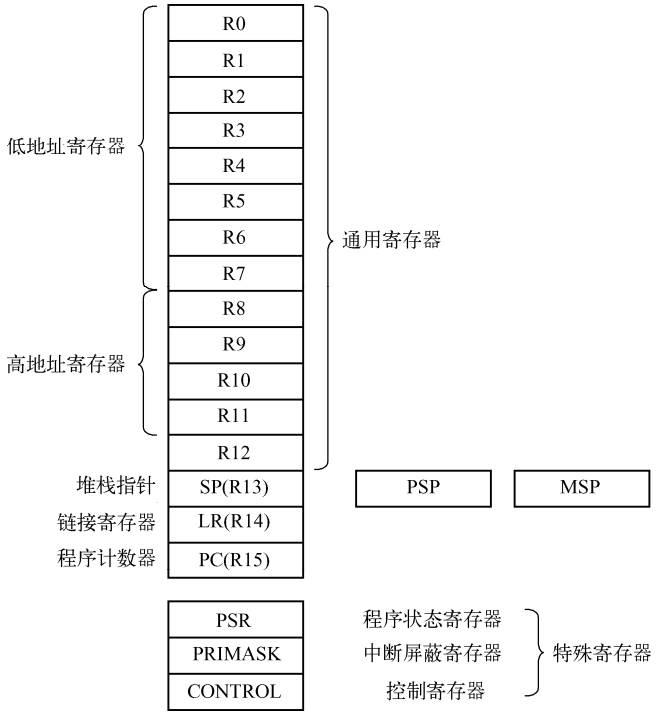


图 1-7 处理器内核寄存器组

（1）通用寄存器

R0～R12 是供数据操作使用的 32 位通用寄存器。

（2）堆栈指针

堆栈指针（SP）是寄存器 R13。在线程模式中，CONTROL 寄存器的 bit[1]指示了堆栈

指针的使用情况。

0：主堆栈指针（MSP）。这是复位值。

1：进程堆栈指针（PSP）。

复位时，处理器将地址 0x00000000 的值加载到 MSP 中。

（3）链接寄存器

链接寄存器（LR）是寄存器 R14。它保存子程序、函数调用和异常的返回信息。复位时，LR 的值不可知。

（4）程序计数器

程序计数器（PC）是寄存器 R15。它包含当前的程序地址。复位时，处理器将复位向量（地址：0x00000004）的值加载到 PC。位 0 的值复位时被加载到 EPSR 的 T 位，必须为 1。

（5）程序状态寄存器

程序状态寄存器（PSR）由下列 3 种寄存器组合而成：

- 应用程序状态寄存器（APSR）；
- 中断程序状态寄存器（IPSR）；
- 执行程序状态寄存器（EPSR）。

在 32 位的 PSR 中，这 3 个寄存器的位域分配互斥。PSR 的位分配见表 1-5。

表 1-5　APSR、IPSR、EPSR 寄存器的位分配

这 3 个寄存器可以单独访问，也可以 2 个一组或 3 个一组进行访问，访问时，将寄存器名称作为 MSR 或 MRS 指令的一个变量。例如：

- 使用寄存器名称 PSR，用 MRS 指令来读所有寄存器。
- 使用寄存器名称 APSR，用 MSR 指令来写 APSR。

PSR 的组合和属性见表 1-6。

表 1-6　PSR 的组合和属性

寄　存　器	类　　型	组　　合
PSR R/W	[1][2]	APSR、EPSE 和 IPSR
IEPSR RO		EPSE 和 IPSR
IAPSR R/W	[1]	APSR 和 IPSR
EAPSR R/W	[2]	APSR 和 EPSR

注：[1] 处理器忽略对 IPSR 位的写操作。

[2] 读 EPSR 位时返回零，处理器忽略对 EPSR 位的写操作。

① 应用程序状态寄存器：APSR 包含执行完前面的指令后条件标志的当前状态。APSR

的位分配见表 1-7。

表 1-7　APSR 的位分配

位	名　称	功　能
[31]	N	负值标志
[30]	Z	零值标志
[29]	C	进位或借位标志
[28]	V	溢出标志
[27:0]	—	保留

② 中断程序状态寄存器：ISP 包含当前中断服务程序（ISR）的异常编号。IPSR 的位分配见表 1-8。

表 1-8　IPSR 的位分配

位	名　称	功　能
[31:6]	—	保留
[5:0]	异常编号	ISR_NUMBER:是当前异常的编号。 0=线程模式 1=保留 2=NMI 3=HardFault 4～10=保留 11 = SVCall 12, 13 = 保留 14 = PendSV 15 = SysTick 16～47 = IRQ0～IRQ31 48～63 = 保留

③ 执行程序状态寄存器：EPSR 包含 Thumb 状态位。EPSR 的位分配见表 1-9。

表 1-9　EPSR 的位分配

位	名　称	功　能
[31:25]		保留
24	T	Thumb状态位
[23:0]	—	保留

如果应用软件使用 MRS 指令直接读取 EPSR 将始终返回零。利用 MSR 指令来写 EPSR 的操作会被忽略。故障处理程序可以检查入栈 PSR 的 EPSR 值来确定故障的原因。下面的操作可以清除 T 为 0。

[1] 指令 BLX、BX 和 POP{PC}。

[2] 异常返回时恢复被压入栈中的 xPSR 值。

[3] 进入异常时向量值的 bit[0]。

在 T 位为 0 时尝试执行指令会导致 HardFault 或锁定故障。

可中断-可重启的指令有 LDM 和 STM。如果在执行这两条中的其中一条指令的过程中出现中断，处理器就终止指令的执行。在处理完中断后，处理器再从头开始重新执行指令。

（6）异常屏蔽寄存器

异常屏蔽寄存器禁止处理器处理异常。当异常可能影响到实时任务或要求连续执行的代码序列时，异常就被禁能。可以使用 MSR 和 MRS 指令或 CPS 指令改变 PRIMASK 的值来禁能或重新使能异常。PRIMASK 寄存器阻止优先级可配置的所有异常被激活。有关寄存器的属性见表 1-10。

表 1-10　PRIMASK 寄存器的位分配

位	描　　述
31:1	保留
0	0：无影响。 1：阻止可配置优先级的异常激活

（7）控制（CONTROL）寄存器

CONTROL 寄存器控制着处理器处于线程模式时所使用的堆栈。CONTROL 寄存器的位分配见表 1-11。

表 1-11　CONTROL 寄存器的位分配

位	功　　能
31:2	保留
CONTROL[1]	堆栈指针选择： 0代表当前堆栈指针是MSP，1代表当前堆栈指针是PSP
[0]	保留

处理器模式始终使用 MSP，因此，在处理器模式下，处理器忽略对 CONTROL 寄存器的有效堆栈指针位执行的明确的写操作。异常进入和返回机制会将 CONTROL 寄存器更新。在一个 OS 环境中，推荐运行在线程模式中的线程使用进程堆栈，内核和异常处理器使用主堆栈。默认情况下，线程模式使用 MSP。要将线程模式中使用的堆栈指针切换成 PSP，只需要使用 MSR 指令将有效堆栈位设置为 1。

注意：当更改堆栈指针时，软件必须在 MSR 指令后立刻使用一个 ISB 指令。这样来保证 ISB 之后的指令执行时使用新的堆栈指针。

1.4　存储器模型

Cortex-M0 支持 32 位字、16 位半字、8 位字节，所有数据存储器访问都采用小端模式。指令存储器和专用外设总线（PPB）访问始终是小端模式。处理器有一个固定的存储器映射，提供 4GB 的可寻址存储空间。存储器映射如图 1-8 所示。

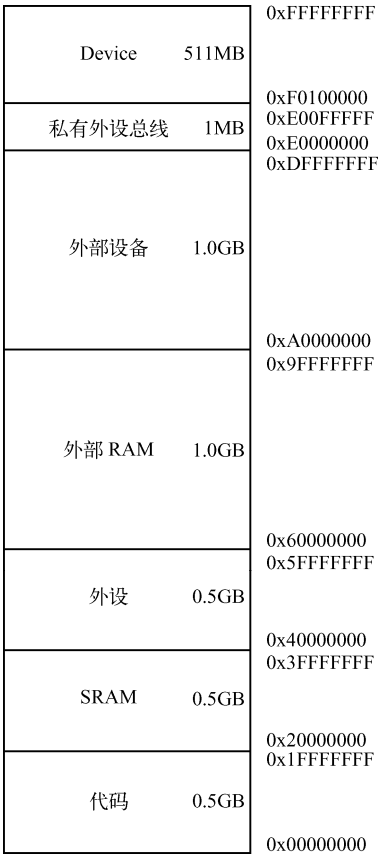


图 1-8　Cortex-M0 存储器映射

1.4.1　存储区、类型和属性

存储器映射分成多个区域。每个区域有一个相应的存储器类型，某些区域还有附加的存储器属性。存储器类型和属性决定了各个区域的访问方式。

主要的存储器类型如下。

- 常规存储器（Normal）：处理器为了提高效率，可以重新对交易进行排序，或者刻意地进行读取。
- Device 存储器：处理器保护与 Device 或强秩序存储器（Strong-ordered memory）的其他交易相关的交易秩序。
- 强秩序存储器（Strong-Ordered）：处理器保护与所有其他交易相关的交易秩序。

Device 存储器和强秩序存储器的不同秩序要求意味着，存储器系统可以缓冲一个对 Device 存储器的写操作，但不准缓冲对强秩序存储器的写操作。

1.4.2　存储器系统的存储器访问次序

对于大多数由明确的存储器访问指令引发的存储器访问，存储器系统都不保证访问秩序与指令的编写顺序完全一致，只要所有访问秩序的重新安排不影响指令序列的操作就行。一

般情况下，如果两个存储器访问的顺序必须与两条存储器访问指令编写的顺序完全一致程序才能正确执行，软件就必须在两条存储器访问指令之间插入一条内存屏障指令。但是，存储器系统不保证 Device 存储器和强秩序存储器的一些访问秩序。对于两条存储器访问指令 A1 和 A2，如果 A1 的编写顺序在前，两条指令所引发的存储器访问顺序见表 1-12。

表 1-12　存储器访问排序

A1　　A2	常规访问	Device 访问		强秩序访问
		非共享	共享	
常规访问	–	–	–	–
Device 访问，非共享	–	<	–	<
Device 访问，非共享	–	–	<	<
强秩序访问	–	<	<	<

注：– 表示存储器系统不保证访问秩序。

< 表示观察到访问顺序与指令编写顺序一致，即 A1 总是在 A2 之前。

1.4.3　存储器访问的行为

存储器映射中每个区域的访问行为见表 1-13。

表 1-13　存储器访问的行为

地 址 范 围	存 储 区 域	存储器类型	XN	描　　述
0x00000000～0x1FFFFFFF	Code	常规存储	—	程序代码的可执行区域。也可以把数据保存到这里
0x20000000～0x3FFFFFFF	SRAM	常规存储器	—	数据的可执行区域。也可以把代码保存到这里
0x40000000～0x5FFFFFFF	外设	Device存储器	XN	外部设备存储器
0x60000000～0x9FFFFFFF	外部RAM	常规存储器	—	数据的可执行区域
0xA0000000～0xDFFFFFFF	外部设备	Device存储器	XN	外部设备存储器
0xE0000000～0xE00FFFFF	专用外设总线	强秩序存储器	XN	这个区域包括NVIC、系统定时器和系统控制块。这个区域只能使用字访问
0xE0100000～0xFFFFFFFF	Device	Device存储器	XN	包含STM32的标准外设

1.4.4　软件的存储器访问顺序

因为如下原因，程序流程的指令顺序不能担保相应的存储器交易顺序。

- ❑ 为了提高效率，处理器可以将一些存储器访问的顺序重新安排，只要不影响指令序列的操作就行；
- ❑ 存储器映射中的存储器或设备可能有不同的等待状态；
- ❑ 某些存储器访问被缓冲，或者是刻意为之。

1.4.2 节描述了存储器系统在哪些情况下能保证存储器访问的秩序。但是，如果存储器访问的秩序十分重要，软件就必须插入一些内存屏障指令来强制保持存储器访问的秩序。处

理器提供了以下内存屏障指令（memory barrier instruction）。

- DMB：数据存储器屏障（DMB）指令保证先完成未处理的存储器交易，再执行后面的存储器交易。
- DSB：数据同步屏障（DSB）指令保证先完成未处理的存储器交易，再执行后面的指令。
- ISB：指令同步屏障（ISB）保证所有已完成的存储器交易的结果后面的指令都能辨认出来。

下面是内存屏障指令使用的一些例子。

- 向量表：如果程序改变了向量表中的一项，然后又使能了相应的异常，那么就在两个操作之间插入一条 DMB 指令。这就确保了，如果异常在被使能后立刻被采纳，处理器能使用新的异常向量。
- 自修改代码：如果一个程序包含自修改代码，代码修改之后在程序中立刻使用一条 ISB 指令。这就确保了后面的指令执行使用的是更新后的程序。
- 存储器映射切换：如果系统包含一个存储器映射切换机制，在切换存储器映射之后使用一条 DSB 指令。这就确保了后面的指令执行使用的是更新后的存储器映射。

对强秩序存储器（例如，系统控制块）执行的存储器访问不需要使用 DMB 指令。处理器保护与所有其他交易相关的交易秩序。

1.5 异 常 模 型

1. 异常状态

每个异常都处于下面其中一种状态。

- 无效：异常无效，未挂起。
- 挂起：异常正在等待处理器处理。一个外设或软件的中断请求可以改变相应的挂起中断的状态。
- 有效：一个异常正在被处理器处理，但处理尚未结束。一个异常处理程序可以中止另一个异常处理程序的执行。在这种情况下，两个异常都处于有效状态。

2. 异常类型

异常类型主要有以下几种。

- 复位：在上电或热复位时启动。异常模型将复位当作一种特殊形式的异常来对待。当复位产生时，处理器的操作停止（可能停止在一条指令的任何一点上）。当复位撤销时，从向量表中复位项提供的地址处重新启动执行。执行在线程模式下重新启动。
- NMI（不可屏蔽中断）：可以由外设产生，也可以由软件来触发。这是除复位之外优先级最高的异常。NMI 永远使能，优先级固定为 2。NMI 不能被屏蔽，它的执行也不能被其他任何异常中止，不能被除复位之外的任何异常抢占。
- HardFault：由于在正常操作过程中或在异常处理过程中出错而出现的一个异常。HardFault 的优先级固定为-1，表明它的优先级要高于任何优先级可配置的异常。

- SVCall：管理程序调用（SVC）异常是一个由 SVC 指令触发的异常。在 OS 环境下，应用程序可以使用 SVC 指令来访问 OS 内核函数和器件驱动。
- PendSV：一个中断驱动的系统级服务请求。在 OS 环境下，当没有其他异常有效时，使用 PendSV 来进行任务切换。
- SysTick：一个系统定时器到达零时产生的异常。软件也可以产生一个 SysTick 异常。在 OS 环境下，处理器可以将这个异常用作系统节拍。
- 中断（IRQ）：外设发出或由软件请求产生的异常。所有中断都与指令执行不同步。在系统中，外设使用中断来与处理器通信。

表 1-14 说明各种异常类型以及向量地址特性。

表 1-14　各种异常类型的特性

异常号[1]	IRQ 号[1]	异常类型	优　先　级	向量地址[2]	活动
1	—	复位	-3，优先级最高	0x00000004	异步
2	-14	NMI	-2	0x00000008	异步
3	-13	HardFault	-1	0x0000000C	同步
4～10	—	保留	—		—
11	-5	SVCall	可配置	0x0000002C	同步
12～13	—	保留	—		—
14	-2	PendSV	可配置	0x00000038	异步
15	-1	SysTick	可配置	0x0000003C	异步
16～47	0～31	中断（IRQ）	可配置	0x00000040 及以上[2]	异步

注：[1]为了简化软件层，CMSIS 只使用 IRQ 编号，因此，对除中断外的其他异常都使用负值。IPSR 返回异常编号。
[2] 地址值以 4 为步长，逐次递增。

对于除复位之外的异步异常，在异常被触发和处理器进入异常处理程序之间，处理器可以执行条件指令。

3. 异常处理程序

处理器使用以下处理程序处理异常。

（1）中断服务程序（ISR）：中断 IRQ0～IRQ31 是由 ISR 来处理的异常。

（2）故障处理程序：HardFault 是唯一一个由故障处理程序来处理的异常。

（3）系统处理程序：NMI、PendSV、SVCall、SysTick 和 HardFault 是由系统处理程序处理的全部异常。

4. 向量表

向量表包含堆栈指针的复位值以及所有向量处理程序的起始地址（也称为异常向量）。图 1-9 显示了异常向量在向量表中的放置顺序。每个向量的最低有效位必须为 1，表明异常处理程序都是用 Thumb 代码编写的。向量表的地址固定为 0x00000000。

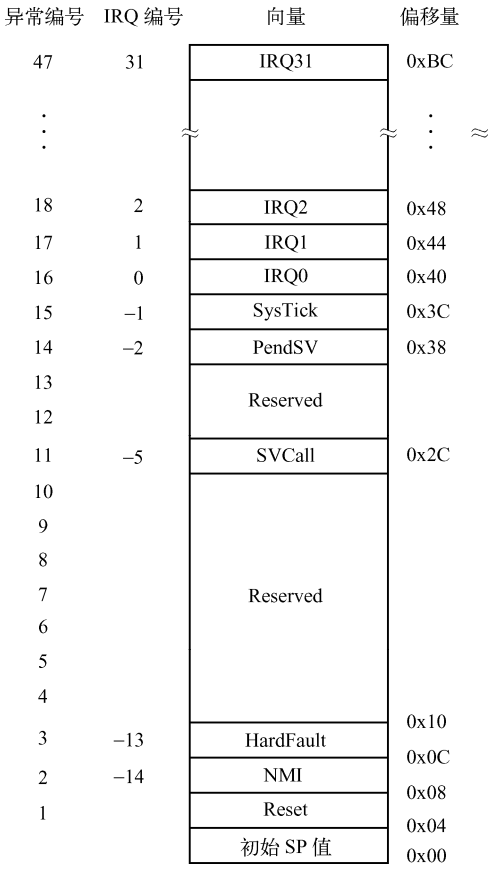


图 1-9　向量表

5. 异常优先级

如表 1-14 所示，每个异常都有对应的优先级，越小的优先级值指示一个更高的优先级；除复位、HardFault 和 NMI 之外，所有异常的优先级都是可配置的；如果软件不配置任何优先级，那么，所有优先级可配置的异常的优先级就都为 0。

优先级值的配置范围为 0～192。复位、HardFault 和 NMI 这些有固定的负优先级值的异常优先级高于任何其他的异常。若给 IRQ[0]分配一个高优先级值，给 IRQ[1]分配一个低优先级值，就意味着 IRQ[1]的优先级高于 IRQ[0]。如果 IRQ[1]和 IRQ[0]都有效，先处理 IRQ[1]。

如果多个挂起的异常具有相同的优先级，异常编号越小的挂起异常优先处理。例如，如果 IRQ[0]和 IRQ[1]正在挂起，并且两者的优先级相同，那么先处理 IRQ[0]。

当处理器正在执行一个异常处理程序时，如果出现一个更高优先级的异常，那么这个异常就被抢占。如果出现的异常的优先级和正在处理的异常的优先级相同，这个异常就不会被抢占，与异常的编号大小无关。但是，新中断的状态为挂起。

6. 异常的进入和返回

有关异常进入与退出涉及下面 4 个定义。

（1）抢占：当处理器正在执行一个异常处理程序时，如果一个异常的优先级比正在处理

的异常的优先级更高，那么低优先级的异常就被抢占。当一个异常抢占另一个异常时，这些异常就被称为嵌套异常。

（2）返回：当异常处理程序结束，并且满足以下条件时，异常就返回。没有优先级足够高的挂起异常要处理；已结束的异常处理程序没有在处理一个迟来的异常；处理器弹出堆栈，处理器状态恢复到中断出现之前的状态。

（3）末尾连锁（Tail-chaining）：这个机制加速了异常的处理。当一个异常处理程序结束时，如果一个挂起的异常满足异常进入的要求，就跳过堆栈弹出，控制权移交给新的异常处理程序。

（4）迟来（Late-arriving）：这个机制加速了抢占的处理。如果一个高优先级的异常在前一个异常正在保存状态的过程中出现，处理器就转去处理更高优先级的异常，开始提取这个异常的向量。状态保存不受迟来异常的影响，因为两个异常保存的状态相同。从迟来异常的异常处理程序返回时，要遵守正常的末尾连锁规则。

异常的进入是指当有一个优先级足够高的挂起异常存在，并且满足下面的任何一个条件，就进入异常处理：

（1）处理器处于线程模式。

（2）新异常的优先级高于正在处理的异常，这时，新异常就抢占了正在处理的异常。当一个异常抢占了另一个异常时，异常就被嵌套。

优先级足够高是指该异常的优先级比屏蔽寄存器中所限制的任何一个异常组的优先级都要高。优先级比这个异常要低的异常被挂起，但不被处理器处理。

当处理器处理异常时，除非异常是一个末尾连锁异常或迟来的异常，否则，处理器都把信息压入到当前的堆栈中。这个操作被称为入栈（stacking），8 个数据字的结构被称为栈帧（stack frame）。图 1-10 是异常进入时堆栈的内容。

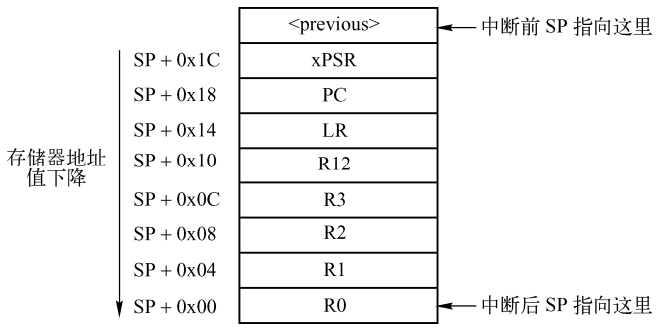


图 1-10　异常进入时堆栈的内容

入栈后，堆栈指针立刻指向栈帧的最低地址单元。栈帧按照双字地址对齐。栈帧包含返回地址。这是被中止的程序中下条指令的地址。这个值在异常返回时返还给 PC，使被中止的程序恢复执行。处理器执行一次向量提取，从向量表中读出异常处理程序的起始地址。当入栈结束时，处理器开始执行异常处理程序。同时，处理器向 LR 写入一个 EXC_RETURN 值。这个值指示了栈帧对应哪个堆栈指针以及在异常出现之前处理器处于什么工作模式。如果在异常进入的过程中没有更高优先级的异常出现，处理器就开始执行异常处理程序，并自动将相应的挂起中断的状态变为有效。如果在异常进入的过程中有另一个优先级更高的异常

出现，处理器就开始执行这个高优先级异常的异常处理程序，不改变前一个异常的挂起状态。这是一个迟来异常的情况。

异常返回是指当处理器处于处理器模式并且执行下面其中一条指令尝试将 PC 设为 EXC_RETURN 值时，出现异常返回。

- POP 指令：用来加载 PC。
- BX 指令：用来使用任意的寄存器。

在异常进入时处理器将一个 EXC_RETURN 值保存到 LR 中。异常机制依靠这个值来检测处理器何时执行完一个异常处理程序。EXC_RETURN 值的 bit[31:4]为 0xFFFFFFF。当处理器将一个相应的这种形式的值加载到 PC 时，它将检测到这个操作并不是一个正常的分支操作，而是异常已经结束。因此，处理器启动异常返回序列。EXC_RETURN 的 bit[3:0]指出了所需的返回堆栈和处理器模式，见表 1-15。

表 1-15　异常返回的行为

EXC_RETURN	描　　述
0xFFFFFFF1	返回到处理器模式：异常返回获得主堆栈的状态。 返回后执行使用 MSP
0xFFFFFFF9	返回到线程模式：异常返回获得 MSP 的状态。返回后执行使用 MSP
0xFFFFFFFD	返回到线程模式：异常返回获得 PSP 的状态。返回后执行使用 PSP
所有其他值	保留

7. 故障处理

故障是异常的一个子集。所有的故障都导致 HardFault 异常被处理，或者，如果故障在 NMI 或 HardFault 处理程序中出现，会导致锁定。发生以下情况会导致故障。

- 在一个优先级等于或高于 SVCall 的地方执行 SVC 指令；
- 在没有调试器的情况下执行 BKPT 指令；
- 在加载或存储时出现一个系统产生的总线错误；
- 执行一个 XN 存储器地址中的指令；
- 从系统产生了一个总线故障的地址单元中执行指令；
- 在提取向量时出现了一个系统产生的总线错误；
- 执行一个未定义的指令；
- 由于 T 位之前被清零而导致不再处于 Thumb 状态的情况下执行一条指令；
- 尝试对一个不对齐的地址执行加载或存储操作。

只有复位和 NMI 可以抢占优先级固定的 HardFault 处理程序。HardFault 可以抢占除复位、NMI 或其他硬故障之外的任何异常。

如果在执行 NMI 或 HardFault 处理程序时出现故障，或者，在一个使用 MSP 的异常返回时出栈的却是 PSR 的时候系统产生一个总线错误，处理器进入一个锁定状态。当处理器处于锁定状态时，它不执行任何指令。在下列情况出现之前，处理器保持锁定状态。

- 复位；
- 调试器将锁定状态终止；
- 出现一个 NMI，以及当前的锁定处于 HardFault 处理程序中。

如果锁定状态出现在 NMI 处理程序中，后面的 NMI 就无法使处理器离开锁定状态。

1.6　电 源 管 理

Cortex-M0 处理器的睡眠模式可以降低功耗，通过 SCR 的 SLEEPDEEP 位选择睡眠模式。睡眠模式包含两种。

- ❑ 睡眠模式：停止处理器时钟，其他系统和外设时钟可能仍旧运行。
- ❑ 深度睡眠模式：绝大多数的 STM32 系统和外设时钟停止。产品级对应停止或者待机模式。

下面讲述进入睡眠模式的机制和唤醒条件。

1. 进入睡眠模式

系统可以产生伪唤醒事件，例如，一个调试操作唤醒处理器。因此，软件必须能够在这样的事件之后使处理器重新回到睡眠模式。程序中可以有一个空闲循环使得器件回到睡眠模式。

（1）等待中断

等待中断指令（WFI）使器件立刻进入睡眠模式。当执行一个 WFI 指令时，处理器停止执行指令，进入睡眠模式。

（2）等待事件

等待事件指令（WFE）根据一个一位的事件寄存器的值来进入睡眠模式。处理器执行一个 WFE 指令时检查事件寄存器的值。

0：处理器停止执行指令，进入睡眠模式。

1：处理器将寄存器的值设为 0，并继续执行指令，不进入睡眠模式。

如果事件寄存器为 1，表明处理器在执行 WFE 指令时不必进入睡眠模式。通常的原因是出现了一个外部事件，或者系统中的另一个处理器已经执行了 SEV 指令。软件不能直接访问这个事件寄存器。

（3）Sleep-on-exit

如果 SCR 的 SLEEPONEXIT 位被设为 1，当处理器完成一个异常处理程序的执行并返回到线程模式时，处理器立刻进入睡眠模式。如果应用只要求处理器在中断出现时运行，就使用这种机制。

2. 从睡眠模式唤醒

处理器的唤醒条件取决于使处理器进入睡眠模式所采用的机制。

（1）从 WFI 或 sleep-on-exit 唤醒

通常，只有当检测到一个优先级足够高的异常导致进入异常处理时，处理器才唤醒。某些嵌入式系统在处理器唤醒之后可能必须先执行系统恢复任务，然后再执行中断处理程序。通过将 PRIMASK 位置位来实现这个操作。如果到来的中断被使能，并且优先级高于当前的异常优先级，处理器就唤醒，但不执行中断处理程序，直至处理器将 PRIMASK 设为 0。

（2）从 WFE 唤醒

如果出现以下情况，处理器被唤醒。

（1）处理器检测到一个优先级足够高的异常导致进入异常处理。

（2）在一个多处理器的系统中，系统中的另一个处理器执行了 SEV 指令。

另外，如果 SCR 的 SEVONPEND 位被置 1，那么任何新的挂起中断都能触发一个事件和唤醒处理器，即使这个中断被禁能，或者这个中断的优先级不够高而导致无法进入异常处理。

3. 电源管理编程提示

ISO/IEC C 语言不能直接产生 WFI、WFE 和 SEV 指令。CMSIS 为这些指令提供了以下内在函数：

```
void __WFE(void)   /* 等待事件                    */
void __WFI(void)   /* 等待中断                    */
void __SEV(void)   /* 发送事件                    */
```

1.7 指 令 集

Cortex-M0 采用 ARM V6-M 构架，只支持 56 条 Thumb 指令。这些指令既兼容 Cortex-M3，又兼容 ARM7 处理器。

在开发 Cortex-M0 处理器应用程序时，用户不需要使用汇编语言。但有时为了优化 C 语言程序，提高代码的执行效率，会用到反汇编。所以为了方便查询，这里给出 Cortex-M0 的指令集，见表 1-16。

表 1-16 Cortex-M0 指令集

助 记 符	操 作 数	描 述	影响标志位
ADCS	{Rd,} Rn, Rm	带进位加	N,Z,C,V
ADD{S}	{Rd,} Rn, <Rm|#imm>	加	N,Z,C,V
ADR	Rd, label	将基于PC相对偏移的地址读到寄存器	—
ANDS	{Rd,} Rn, Rm	位与计算	N,Z
ASRS	{Rd,} Rm, <Rs|#imm>	算术右移	N,Z,C
B{cc}	label	跳转{有条件}	—
BICS	{Rd,} Rn, Rm	位清除	N,Z
BKPT	#imm	断点	—
BL	label	带链接的分支跳转	—
BLX	Rm	带链接的间接跳转	—
BX	Rm	间接跳转	—
CMN	Rn, Rm	比较负值	N,Z,C,V
CMP	Rn, <Rm|#imm>	比较	N,Z,C,V
CPSID	i	修改处理器状态，禁止中断	—

续表

助　记　符	操　作　数	描　　述	影响标志位
CPSIE	i	修改处理器状态，使能中断	—
DMB	—	数据存储隔离	—
DSB	—	数据同步隔离	—
EORS	{Rd,} Rn, Rm	异或	N,Z
ISB	—	指令同步隔离	—
LDM	Rn{!}, reglist	加载多个寄存器，访问之后会递增地址	—
LDR	Rt, label	从基于PC相对偏移地址上加载寄存器	—
LDR	Rt, [Rn, <Rm|#imm>]	以字加载寄存器	—
LDRB	Rt, [Rn, <Rm|#imm>]	以字节加载寄存器	—
LDRH	Rt, [Rn, <Rm|#imm>]	以半字加载寄存器	—
LDRSB	Rt, [Rn, <Rm|#imm>]	以有符号字节加载寄存器	—
LDRSH	Rt, [Rn, <Rm|#imm>]	以有符号半字加载寄存器	—
LSLS	{Rd,} Rn, <Rs|#imm>	逻辑左移	N,Z,C
LSRS	{Rd,} Rn, <Rs|#imm>	逻辑右移	N,Z,C
MOV{S}	Rd, Rm	传送Rd数据到Rm	N,Z
MRS	Rd, spec_reg	将特殊功能寄存器内容移到通用寄存器	—
MSR	spec_reg, Rm	将通用寄存器内容移到特殊功能寄存器	N,Z,C,V
MULS	Rd, Rn, Rm	乘法，32结果	N,Z
MVNS	Rd, Rm	取反的位操作	N,Z
NOP	—	空操作	—
ORRS	{Rd,} Rn, Rm	逻辑或	N,Z
POP	reglist	寄存器出栈	—
PUSH	reglist	寄存器压栈	—
REV	Rd, Rm	按字节反转	—
REV16	Rd, Rm	按半字反转	—
REVSH	Rd, Rm	按有符号半字反转	—
RORS	{Rd,} Rn, Rs	循环右移	N,Z,C
RSBS	{Rd,} Rn, #0	逆向减法	N,Z,C,V
SBCS	{Rd,} Rn, Rm	带符号减	N,Z,C,V
SEV	—	发送事件	—
STM	Rn!, reglist	批量存储寄存器，Rn递减	—
STR	Rt, [Rn, <Rm|#imm>]	将寄存器作为字来存储	—
STRB	Rt, [Rn, <Rm|#imm>]	将寄存器作为字节来存储	—

续表

助 记 符	操 作 数	描 述	影响标志位
STRH	Rt, [Rn, <Rm|#imm>]	将寄存器作为半字来存储	—
SUB{S}	{Rd,} Rn, <Rm|#imm>	减法	N,Z,C,V
SVC	#imm	管理调用	—
SXTB	Rd, Rm	字节符号扩展到32位	—
SXTH	Rd, Rm	半字符号扩展到32位	—
TST	Rn, Rm	逻辑与测试	N,Z
UXTB	Rd, Rm	字节零扩展到32位	—
UXTH	Rd, Rm	半字零扩展到32位	—
WFE	—	等待事件	—
WFI	—	等待中断	—

ISO/IEC C 代码不能直接访问 Cortex-M0 指令，而 ARM 公司提供的 CMSIS 可以直接访问 Cortex-M0 指令，并被多个编译器支持，也可以通过编译器的嵌入汇编指令方式进行。表 1-17 是 CMSIS 对应 Cortex-M0 指令的内部函数。

表 1-17 CMSIS 内部函数用于生成 Cortex-M0 指令

指 令	CMSIS intrinsic function
CPSIE i	void enable_irq(void)
CPSID i	void disable_irq(void)
ISB	void ISB(void)
DSB	void DSB(void)
DMB	void DMB(void)
NOP	void NOP(void)
REV	uint32_t REV(uint32_t int value)
REV16	uint32_t REV16(uint32_t int value)
REVSH	uint32_t REVSH(uint32_t int value)
SEV	void SEV(void)
WFE	void WFE(void)
WFI	void WFI(void)

CMSIS 也提供了一些用于通过 MRS 和 MSR 指令访问特殊寄存器的函数，见表 1-18。

表 1-18 CMSIS 的内部函数用于访问特殊寄存器

特殊寄存器	访问方式	CMSIS 函数
PRIMASK	读	uint32_t get_PRIMASK (void)
	写	void set_PRIMASK (uint32_t value)
CONTROL	读	uint32_t get_CONTROL (void)
	写	void set_CONTROL (uint32_t value)
MSP	读	uint32_t get_MSP (void)
	写	void set_MSP (uint32_t TopOfMainStack)
PSP	读	uint32_t get_PSP (void)
	写	void set_PSP (uint32_t TopOfProcStack)

1.8　Cortex-M0 内核外设

Cortex-M0 内核外设如下。

- NVIC：一个嵌入式中断控制器，支持低延迟的中断处理。
- 系统控制块（SCB）：到处理器的编程模型接口。它提供系统执行信息和系统控制，包括配置、控制和系统异常的报告。
- 系统定时器（SysTick）：一个 24 位的递减定时器。可以将其用作一个实时操作系统（RTOS）的节拍定时器，或者用作一个简单的计数器。

1.9　STM32F0 系列

意法半导体（STMicroelectronics，ST）是全球领先的半导体解决方案供应商，为传感及功率技术和多媒体融合应用领域提供创新的解决方案。ST 推出了优秀的 STM32F0 系列产品。

STM32F0 瞄准的正是成本敏感的 8 位和 16 位机应用市场，延续 STM32 的特性，而且继承了 STM32 成熟的 IP 组合，为客户提供完整的产品组合、承诺的生产能力和有竞争力的预算价格。STM32F0 系列产品新增 7 个定时器，这些定时器可控制加热器或电动机等设备，例如，在一个电磁炉内同时控制多个电子元器件。

STM32F0 系列集成了支持 HDMI 接口的消费电子控制器（CEC）的硬件，这有助于简化微控制器在众多家庭多媒体设备内的设计，准许开发人员采用最新的工业标准通信协议来设计接口，同时还能降低微控制器的 CPU、存储器、外设接口的负荷，释放更多的能力去处理其他任务。CEC 可由一个低速 32kHz 外部时钟源驱动，或者为降低系统成本，还可用 8MHz 内部时钟源驱动。此外，STM32F0 的 12MHz I/O 信号状态翻转速度让开发人员能够在更低成本和功耗平台上实现复杂控制算法。

目前 STM32F0 系列主要有 STM32F030 超值型、STM32F0x1、STM32F0x2、STM32F0x 四种形式。

STM32F030 超值系列基于 48MHz 的 ARM Cortex-M0 处理器内核。STM32F030 虽然只有 8 位微控制器的价格，但性能和特性并没有受到任何影响。

STM32F030 采用 ARM Cortex 内核，运算速度高达 48MHz。STM32F030 是 STM32 系列中价格最低的产品，具有全套外设，例如高速 12 位 ADC、先进且灵活的定时器、日历 RTC 和通信接口（如 I²C、USART 和 SPI）。

当然对于用户而言，对于能够满足需求情况，更重要的是价格。现在意法半导体喊出的口号是 32 美分的芯片。

STM32F030 的内部实时时钟和 5 通道直接存储器存取控制器（DMA），是同一价位微控制器所不具备的功能，这两项重要特性有助于简化应用开发，提高产品性能。

意法半导体还推出一套叫做探索套件的 STM32F0 专用评估板，这个探索板配备按键和 LED 指示灯，可通过 USB 数据线直接连接 PC，板子上预装了大量的演示软件和固件例程。探索套件兼容市场领先的第三方软件开发工具提供商 Atollic、IAR、Keil 和 TASKING 开发

的 STM32 软件开发环境。

作为同为 ST 公司的两兄弟，STM8 与 STM32F0 不可避免地会短兵相见，它们在外设功能方面太相似了。表 1-19 是对双方功能统计，表面上看目前 STM32F0 外设方面仅仅多了 USB，但实际上 ST 公司对 STM32F0 的外设功能作了很多改善（例如，STM32F0 相比以往的 ST 公司产品线，RTC 精确到亚秒级）。

表 1-19　STM8 与 STM32F0 对比

芯片	字长	供电	时钟频率	支持 EEPROM	价格	调试接口	低功耗模式	指令集
STM8	8位	最高可到5V	24MHz	支持	占优势	1线（SWIM）	低速、时间等待、暂停、退出暂停	CISC
STM32F0	32位	最高3.6V	48MHz	不支持	相比 STM8 稍贵	两线	低速、睡眠（事件等待）、退出睡眠、深度睡眠	RISC

STM8 的供电电压是 5V，优点是噪声容限高。例如，如干扰是 0.1V 幅值，对 5V 的 ADC 影响是 2%，但对于 3.3V 的 ADC 的影响是 3.03%。STM32F0 的低电压供电最大优点就是功耗低，另外就是使得元件低压化，安全性提高。

目前，通过淘宝网上的 STM8 与 STM32F0 价格对比，STM8 占优势，但后续情况未知，毕竟 STM32F0 是 ST 公司用来抢占单片机市场（理论上也包含 STM8 的市场）的，而且 Cortex-M0 也是 ARM 公司用来抢占低端 CPU 市场的。对于芯片公司，量产与客户群对价格影响远大于芯片本身性能的影响。历史上曾出现太多次低性能芯片价高的局面（原因是已经投产的产品，升级 CPU 需研发周期、财力投入。供不应求现象引起了低性能芯片价格虚高）。

SMT32F0 一个很大特点是兼容。不仅仅通过固件库兼容了本系列产品，而且常见外设功能与 STM32F1、STM32F2 等系列产品相同，程序不用修改。但 STM8 在这方面就无向上拓展空间。

如果是规划新产品，建议考虑 STM32F0。毕竟目前是 ST 公司主推的低端 CPU，而且有 ARM 公司做强大的后盾支持。

而且 ST 公司是一家不断完善自己产品的公司，即发布晚的产品线会对以往的产品功能有所改善。例如，STM32F0 的 USART 外设增加的 Modbus 协议硬件支持，能够解决 Modbus 的 RTU 超时检测与 Mobus 的 ASCII 的字符匹配功能。该功能在 STM8 是没有的，即使 STM32F1、STM32F2 系列也是不包含的。从这点上看，选择 STM32F0 就优于 STM8 产品。

1.10　小　结

本章介绍了 Cortex-M0 的特点和其与 8 位机相比的优势；从 8 位机过渡到 Cortex-M0 的注意事项；分析了 Cortex-M0 的构架结构以及原理；最后给出了 STM32F0 的特点及优势，并通过与 STM8 的一些对比，说明了选用 STM32F0 的益处。

第 2 章

开发软件准备

目前第三方软件开发工具提供商 Atollic、IAR、Keil 和 TASKING 的软件开发环境均支持 STM Cortex-M0 的开发。

Atollic TrueSTUDIO 采用 Eclipse™集成开发环境（IDE）框架，拥有 ARM 处理器专用的 GNU 编译器/调试器。免费的 Lite 版开发工具包括预编译运行时库，有 32Kb 的代码限制（对于 Cortex-M0 与 Cortex-M1 是 8Kb 的代码限制）。读者可从 http://www.atollic.com/下载 Atollic TrueSTUDIO。

Tasking（http://www.tasking.com/）提供 32 位与 8 位/16 位的编译环境，可支持 8051、196、ARM、DSP56xx、Power Architecture。2001 年被 Altium 公司收购。

另外一个值得推荐的是 CooCox 的集成开发环境，隶属于派睿集团。该软件属于 IDE，需要额外配置 GCC 编译器。该软件目前支持 Colink、J-Link、ST-Link 等仿真器。

目前国内的 8051 用户普遍使用 Keil C51，所以本章将讲解 Keil 家族中 MDK-ARM 开发环境。

2.1 MDK-ARM 开发环境

μVision 是 Keil 公司的一个集成开发环境（IDE），与 Eclipse 类似。它包括工程管理、源代码编辑、编译设置、下载调试和模拟仿真等功能。μVision 被包含在 Keil 的各种开发工具中，例如 MDK、C51、C166 等。本节以常用的 μVision 4 为例进行讲解。

2.1.1 μVision4 IDE 概述

Keil C51 是针对 51 系列的开发工具，MDK-ARM 是 Keil 公司针对 ARM 的开发工具。二者相互独立，但均采用了 μVision 集成开发环境。μVision4 IDE 是一个窗口化的软件开发平台，它集成了功能强大的编辑器、工程管理器和各种编译工具（包括 C 编译器、宏汇编器、链接/装载器和十六进制文件转换器）。

注：CooCox 的 CoIDE 实际上也属于一款 IDE，不包含任何编译环境。早期 CooCox 的 CoIDE 可看到 Eclipse 的一些信息。μVision 与 Eclipse 类似，Keil C51 与 MDK-ARM 共用这个 IDE 平台。

μVision4 包含以下功能组件，能加速嵌入式应用程序开发过程。

- 功能强大的源代码编辑器；
- 可根据开发工具配置的设备数据库；
- 用于创建和维护工程的工程管理器；
- 集汇编、编译和链接过程于一体的编译工具；
- 用于设置开发工具配置的对话框；
- 真正集成高速 CPU 及片上外设模拟器的源码级调试器；
- 高级 GDI 接口，可用于目标硬件的软件调试和 Keil U-Link 仿真器的连接；
- 用于下载应用程序到 Flash ROM 中的 Flash 编程器；
- 完善的开发工具手册、设备数据手册和用户向导。

μVision4 IDE 使用简单、功能强大，是设计者完成设计任务的重要保证。μVision4 IDE 还提供了大量的例程及相关信息，有助于开发人员快速开发嵌入式应用程序。μVision4 IDE 提供了 Build Mode（编译）和 Debug Mode（调试）两种工作模式。编译模式用于维护工程文件和生成应用程序；调试模式下，既可以用功能强大的 CUP 和外设仿真器测试程序，也可以使用调试器经 Keil U-Link 适配器（或其他驱动器）连接目标系统来测试程序。表 2-1 是 μVision4 的特征。

表 2-1　μVision4 的特征

特　　性	优　　点
μVision4模拟器是模拟多数片内外设的唯一调试器	可在产品硬件准备就绪之前编写代码和测试代码，并可对多种硬件进行调研优化硬件设计
System Viewer可用于监视外设内容	可定义自己的view来监视外设。可在System Viewer窗口观察到详细的状态信息，并可直接修改
μVision4模拟器中的代码覆盖率功能提供了程序执行情况的统计数据	安全性要求的系统可以被全面测试
逻辑分析仪以图形化的方式显示了变量变化情况	以图形方式研究信号和变量的变化，观察它们的相关性
μVision4设备数据库自动配置了目标控制器的开发工具	设备设置可降低开发者的配置工具时间
Template editor方便插入相同的文本序列或者头文件块	使用模板可插入标准文本、头文件描述和通用代码模块
浏览功能可在不同代码块之间快速移动	开发时节约时间
配置向导提供了图形化维护设备和启动代码的配置	替代原有的浏览启动文件进行修改的方式，现在可使用高级GUI特征
目标调试器和模拟器接口可实际模拟外设	极大降低学习曲线

2.1.2　编译、调试现有 MDK 工程

先以 ST 公司提供的 Discovery 中的例程，演示如何使用 MDK-ARM。首先从 ST 网下载 STM32F030、STM32F051、STM32F072 的任何一款芯片对应的 Discovery 软件包并解压。

（1）打开 MDK-ARM μVision4 IDE，如图 2-1 所示。

（2）在 Project 菜单，选择 Open Project…命令，在弹出的 Select Project File 对话框中打开 Project\Master_Workspace\MDK-ARM 目录中的工程文件（STM32F030、STM32F05X、STM32F072X 系列对应文件名称稍微不同，其中 STM32F072 对应的名称为 STM32F072B-Discovery.uvmpw）。单击 Open 按钮，在 Project 窗口打开所选文件。可通过 Project 菜单，选

择 Batch Build，可编译所有工程。编译完成后，在 Build Output 窗口显示编译成功信息，如图 2-2 所示。

图 2-1　MDK-ARM µVision4 IDE 环境

图 2-2　编译结果

如果相应的 Discovery 连接到 PC，则通过 start/Stop Debug Session 可将代码写入并开始调试过程。

2.1.3　创建一个 Keil 新项目

选择 Project→New Project 菜单（见图 2-3），µVision4 将弹出一个标准对话框，可输入希望创建的工程名字，即创建一个新的工程。这里输入 HelloWorld，µVision4 工程文件的后缀为“.uvproj”，然后单击“保存”按钮。

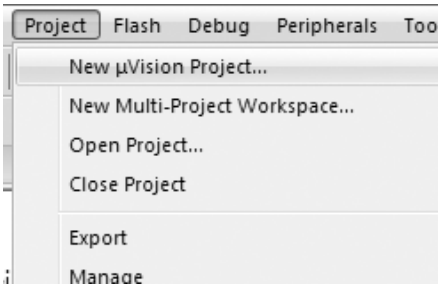


图 2-3　创建新项目

这时会弹出一个对话框要求你选择目标设备的型号，如图 2-4 所示。

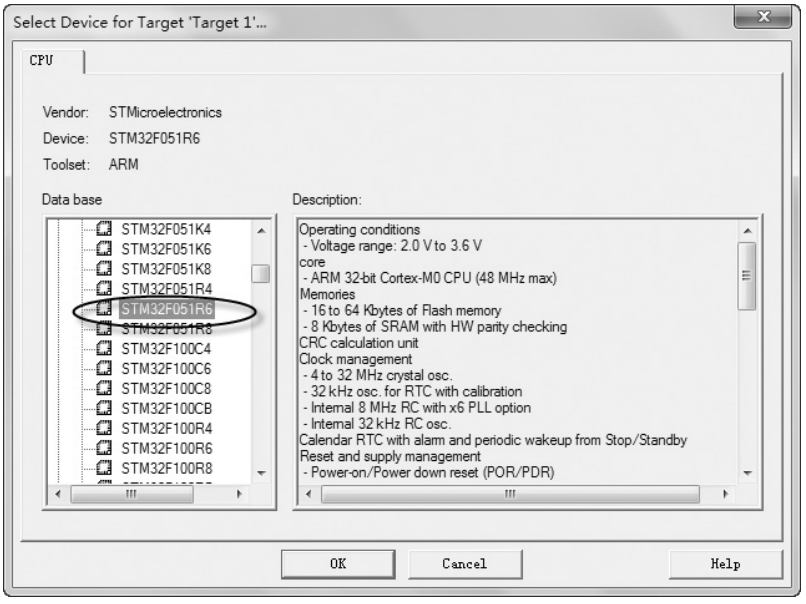


图 2-4　设备选择对话框

低版本的 MDK 无 STM32F0 的设备数据，本图是 MDK4.73 的设备数据。对于 STM32F030 系列，可通过http://www.keil.com/download/files/mdk-arm_addon_stm32f030.zip 下载 MDK 针对 STM32F030系列芯片插件。安装完毕后，选择设备对话框时会弹出图 2-5 所示界面。

在图 2-4 中选择 STM32F051R6。在图 2-4 的右边是关于 STM32F051R6 的一些描述，主要有工作环境、Cortex-M0 内核、片内存储器、电源管理、复位和供电管理、5 通道 DMA 控制器、1 个 12 位 D/A 转换器、高达 55 个快速 I/O、11 个定时器、通信接口、SWD 的一些信息说明。

单击“确定”按钮后显示是否复制启动代码（见图 2-6），我们这个时候单击“Yes”按钮，到此一个工程就建好了。

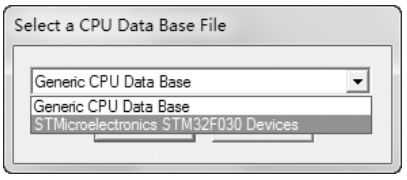


图 2-5　STM32F030 设备选择

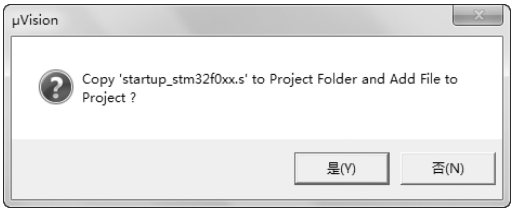


图 2-6　启动代码复制

工程建立好了之后，还要对工程进行进一步的设置，以满足要求。首先用鼠标右键单击左边工程窗口的“Target 1”，会出现一个菜单，选择“Options for Target 'Target 1'”（也可以通过单击工程窗口的“Target 1”，然后使用菜单 Project→Options for Target 'Target 1'），即出现工程配置的对话框，如图 2-7 所示。

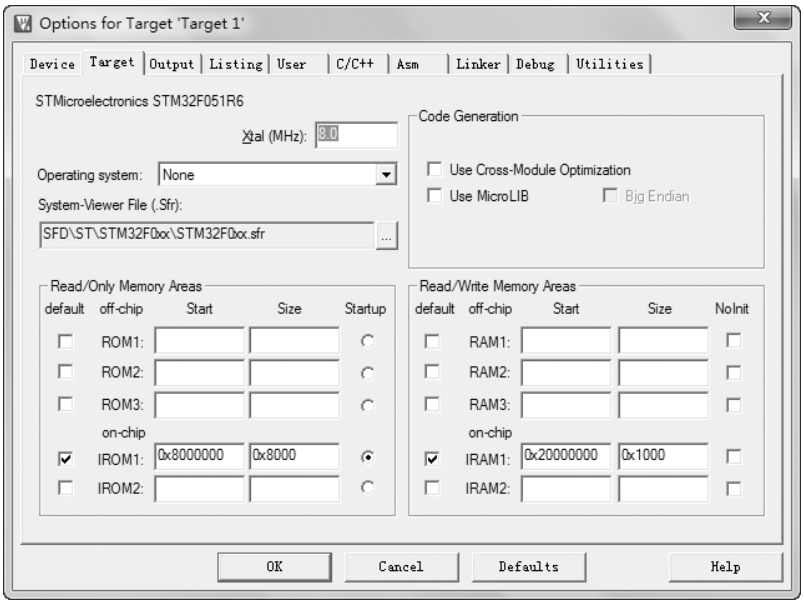


图 2-7　工程配置界面

下面将对图 2-7 中的其他选项配置做一些介绍。

（1）Output 标签页的设置

图 2-8 是编译输出的配置，相应选项说明如下。

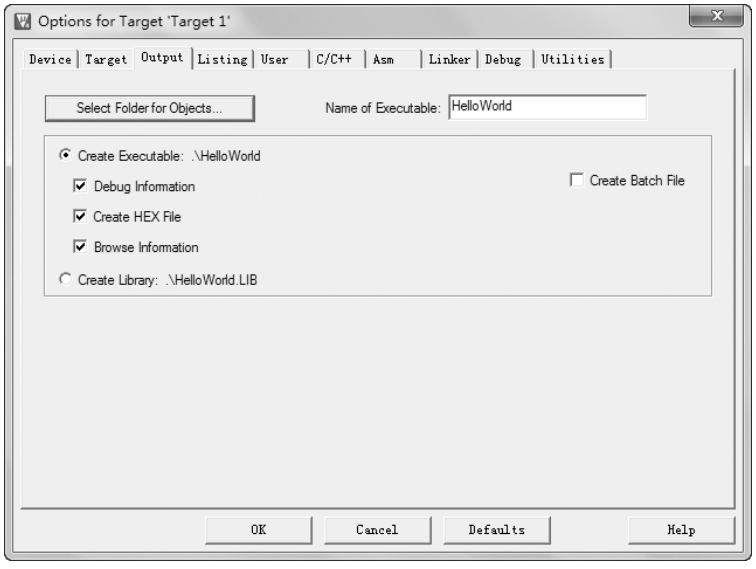


图 2-8　编译输出配置

- Select Folder for Objects：选择编译之后的目标文件存储在哪个目录里。默认位置为工程文件的目录里。
- Name of Executable：生成的目标文件的名字，默认是工程的名字。

- Create Executable：生成可执行程序。
- Debug Information：用于 Debug 版本，生成调试信息，否则的话无法进行单步调试。
- Create Batch File：生成编译的脚本。
- Browse Information：生成源代码浏览信息。
- Create Hex File：这个选项默认情况下未被选中，如果要写片做硬件实验就必须选中该项。
- Create Library：生成 lib 库文件，默认不选。

为了能够浏览源代码以及产品化时烧录程序，常做三处改动。通过 Select Folder for Objects，工程目录下新建了一个 Output 目录保存目标文件，避免和源文件混在一起。另外需选中 Create Hex File 和 Browse Information。

（2）C/C++标签页的设置（见图 2-9）

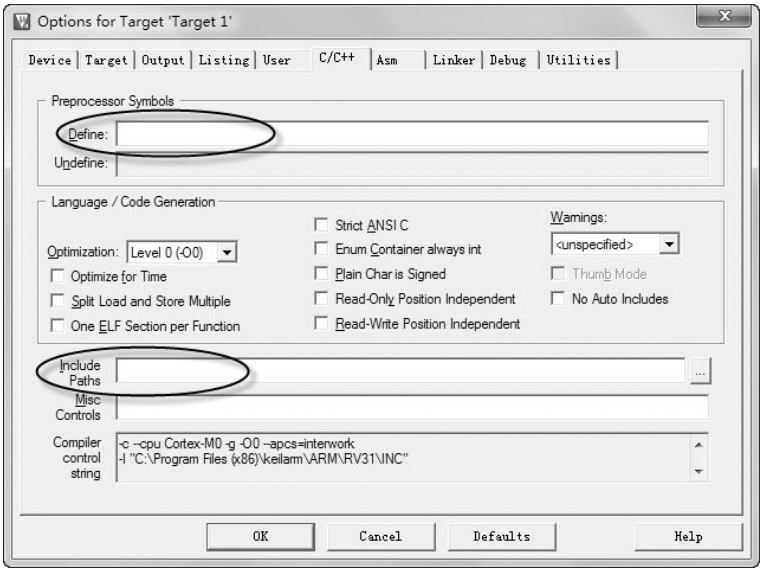


图 2-9　C/C++标签页设置

- Include Paths：指定头文件的查找路径，可以添加多个并用分号隔开。例如：对于采用 STM32F0 Discovery 为模板的开发，此处内容则为..\;..\..\..\..\Libraries\CMSIS\Include;..\..\..\..\Libraries\CMSIS\ST\STM32F0xx\Include;..\..\..\..\Libraries\STM32F0xx_StdPeriph_Driver\inc;..\..\..\..\Utilities\STM32F0-Discovery。
- Define：是预定义的宏。例如：采用 STM32F0 Discovery 开发，此处则为 USE_STDPERIPH_DRIVER STM32F0xx。

（3）Debug 标签页的设置

图 2-10 中左边是对应 μVision4 的模拟环境，右边是针对仿真器，根据仿真器类型进行选择。配置完仿真器类型，需要通过类型旁的 Setting 按钮进入仿真器设置相应的端口类型，针对 STM32F0 均需要设置成 SWD 形式，如图 2-11 所示。并且需要单击图 2-11 中 Flash Download 标签配置 Flash 算法，如图 2-12 所示。

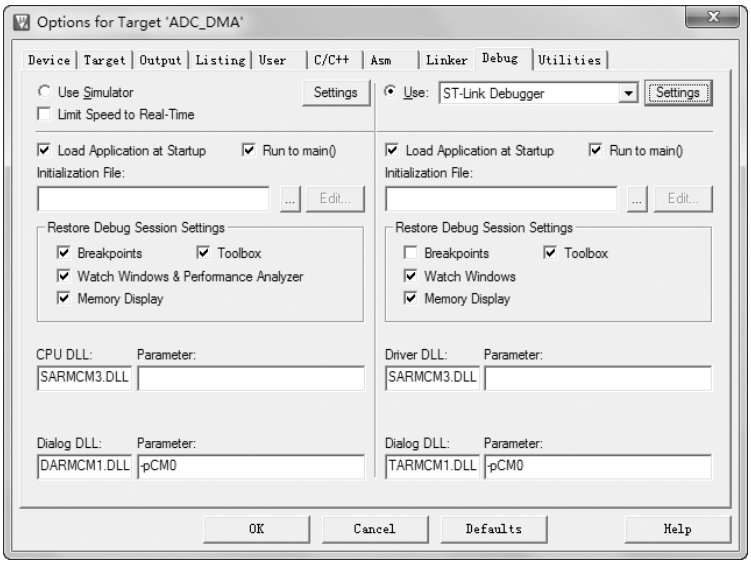


图 2-10　仿真器选择

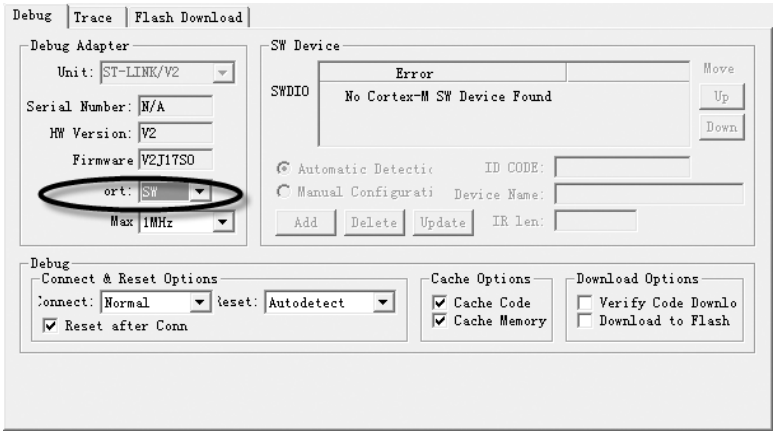


图 2-11　配置 SWD

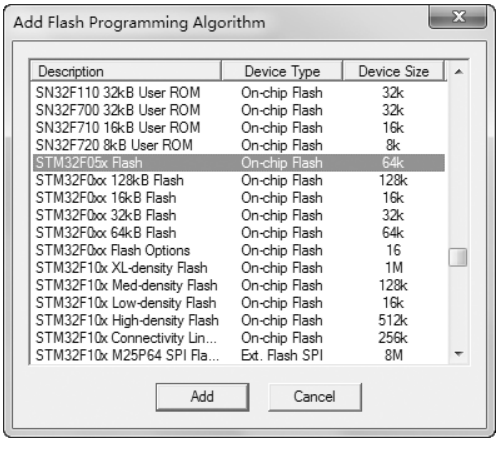


图 2-12　配置 Flash 算法

（4）Utilities 标签页的设置

图 2-13 中的 Use Target Driver for Flash Programming 配置选择 Use Debug Driver，即将 Flash 编程器配置成 Debug 中一致。如果是低版本的 MDK，在 Setting 选项中选择与 Debug 中的 Setting 选项配置相同即可。

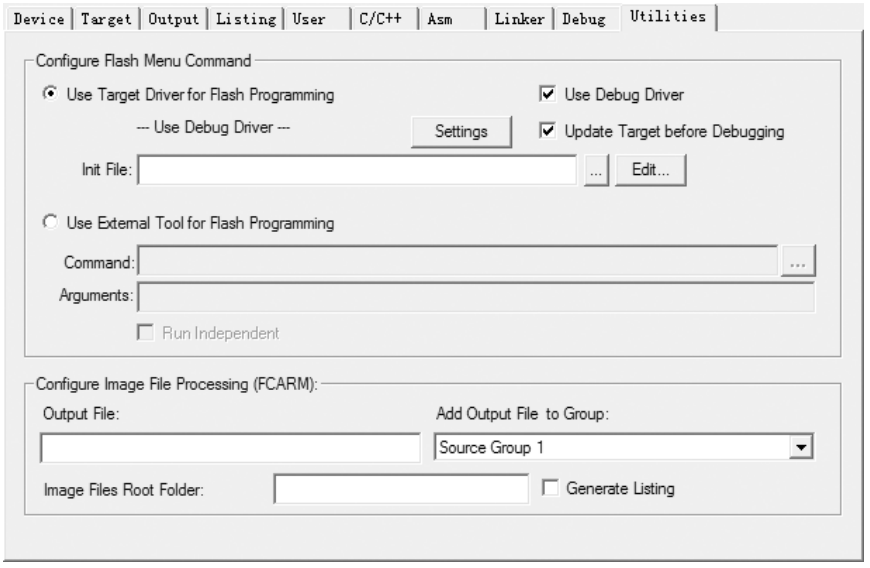


图 2-13　Utilities 设置

选择菜单“File”→“New”或者单击工具栏的新建文件按钮，即可在项目窗口的右侧打开一个新的文本编辑窗口，在该窗口可以输入程序代码。

```
#include <stm32f0xx.h>
void IOInit(void)
{
 RCC -> AHBENR |= 1 << 19;//使能外设 PORTC 时钟
 GPIOC -> MODER &=   0xFFFCFFFF;
 GPIOC -> MODER |=   0x00010000; //PC8 通用输出模式
 GPIOC -> OTYPER &=   0xFFFFFEFF; //PC8 推挽输出
 GPIOC -> OSPEEDR &= 0xFFFCFFFF;
 GPIOC -> OSPEEDR |= 0x00030000; //PC8 高速输出模式
 GPIOC -> ODR |= 1 << 8; //PC8 输出高电平
 }
void IOToggle(void)
{
 int sta;
 //读取 LED1 所在口线的状态，给 sta
 if(GPIOC -> ODR & (1 << 8)) sta = 1;
 else sta = 0;
 if(1 - sta) GPIOC -> BSRR = 1 << 8; //切换 IO 状态
 else   GPIOC -> BRR = 1 << 8;
```

```
}
int main(void)
{int i;
 IOInit();
   while (1)
   {
      for ( i =0;i<100000;i++);
      IOToggle ();
   }
}
```

代码编辑完成之后，我们可以保存源文件，选择菜单“File”→“Save”或者单击工具栏的“保存”按钮，可以用来保存源文件。这时会出现一个保存文件的文件对话框，选择要保存的路径，输入文件名 Hello.c。注意一定要输入扩展名，如果是 c 程序文件，扩展名为.c，如果是汇编文件，扩展名为.s。源文件编辑完成之后我们还需要将源文件加入工程中，工程建好之后，在工程窗口的文件页中，将会出现“Target 1”，前面有个“+”号，单击“+”号展开，可以看到下一层的“Source Group 1”，我们需要向这个里面加入源文件，单击“Source Group 1”使其反白显示，然后单击鼠标右键，出现一个下拉菜单，如图 2-14 所示。

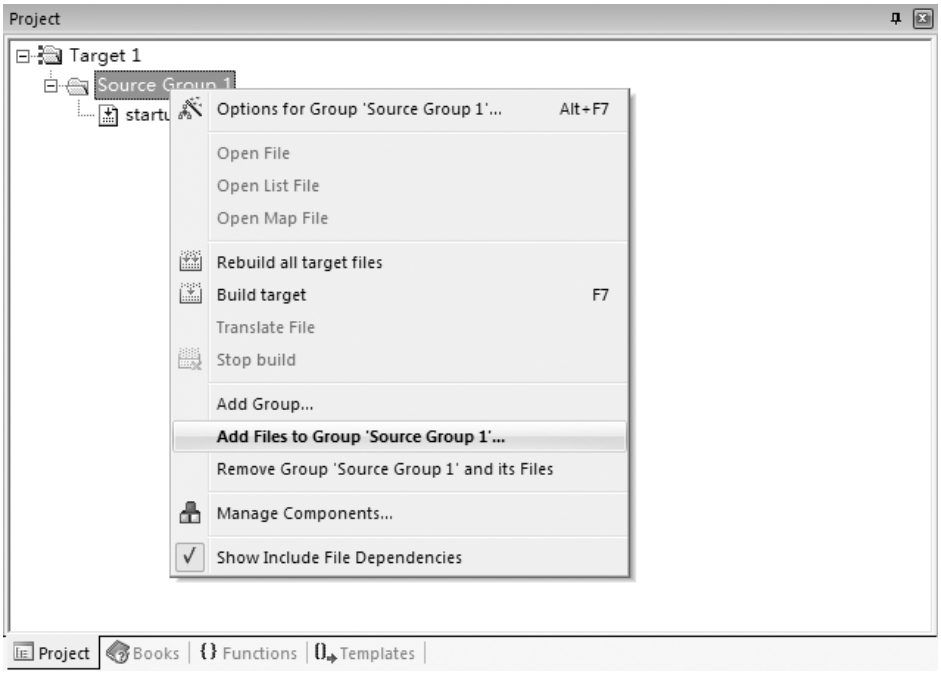


图 2-14　添加文件

选中其中的“Add file to Group 'Source Group 1'”，出现要求寻找源文件对话框，如图 2-15 所示。

图 2-15　源文件选择对话框

注意：该对话框下面的“文件类型”默认为 C source file(*.c)。我们可以通过调整这个来选择过滤我们想要格式的文件，从而帮助我们快速查找文件。如果是汇编文件，就选择“asm source file”；如果是目标文件，选择“Object file”；如果是库文件，选择“Library file”。最后单击“Add”按钮，也可以双击要添加的文件进行加入。

按照添加 hello.c 文件的过程，添加 MDK 安装目录下的\ARM\Startup\ST\STM32F0xx\system_stm32f0xx.c。

通过 Project 菜单，可看到如下有关编译选项。

- Clean target：清除编译结果。
- Build target：编译被修改的文件并且编译应用程序。
- Rebuild all target files：重新编译所有的源文件并且编译应用程序。
- Batch Build：通过前面输出的批处理文件进行编译。
- Translate **.*：编译某个源文件。**.*代表要编译的源文件。
- Stop build：只有编译进行过程中这一项才有效。

在对工程成功进行汇编、连接之后，按 Ctrl+F5 键或者使用菜单“Debug”→“Start/Stop Debug Session”即可进入调试状态。进入调试状态后，界面与编辑状态相比有明显的变化，Debug 菜单项中原来不能用的命令现在已经可以使用了，工具栏会多出一个用于运行和调试的工具栏，如图 2-16 所示。Debug 菜单上的大部分命令可以在此找到对应的快捷按钮。

- Start/Stop Debug Session：开始或者停止调试。
- Run：一直执行下一个活动的断点。
- Step：单步执行。
- Step Over：过程单步执行，即将一个函数作为一个语句来执行。
- Step out of current Function：跳出当前的函数。
- Run to Cursor line：执行到光标所在的行。
- Stop Running：停止运行。

- Breakpoints：打开断点对话框。
- Insert/Remove Breakpoint：在当前行插入/删除一个断点。
- Enable/Disable Breakpoint：激活当前行的断点或者使断点无效。
- Disable All Breakpoints：使程序中所有的断点都无效。
- Kill all Breakpoints：删除程序中所有的断点。

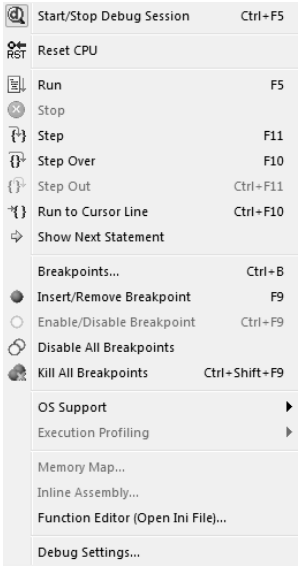


图 2-16 调试菜单命令

通过断点设置可以观察执行结果。断点设置的方法有多种，常用的是在某一程序行设置断点，设置好断点之后可以全速运行程序，一旦执行到该程序行即停止，可在此观察有关的变量值，以确定问题所在。一旦某一行被设置了断点，可以在程序行的左端看到一个红色方框（见图 2-17 调试窗口图），如果该断点被禁用，方框将会变为白色。

```
1
2   #include <stm32f0xx.h>
3   void IOInit(void)
4   {
5     RCC -> AHBENR |= 1 << 19;//使能外设PORTC时钟
6     GPIOC -> MODER &=  0xFFFCFFFF;
7     GPIOC -> MODER |=  0x00010000; //PC8通用输出模式
8     GPIOC -> OTYPER &=  0xFFFFFEFF; //PC8推挽输出
9     GPIOC -> OSPEEDR &= 0xFFFCFFFF;
10    GPIOC -> OSPEEDR |= 0x00030000; //PC8高速输出模式
11    GPIOC -> ODR |= 1 << 8; //PC8输出高电平
12  }
13  void IOToggle(void)
14  {
15    int sta;
16    //读取LED1所在口线的状态,给sta
17    if(GPIOC -> ODR & (1 << 8)) sta = 1;
18    else sta = 0;
19    if(1 - sta) GPIOC -> BSRR = 1 << 8; //切换IO状态
20    else  GPIOC -> BRR = 1 << 8;
21  }
22  int main(void)
23  {int i;
24    IOInit();
25    while (1)
26    {
```

图 2-17　断点设置

调试窗口就是用于查看程序执行状态的。常用窗口中又有存储器窗口（Memory）、观察窗口（Watch）、查看和调用堆栈窗口（Call Stack+Locals）。

存储器窗口可以显示系统中各种内存中的值，常用于检查数组以及某些连续区域内存内容。在 Address 后的编辑框中 0x 表示十六进制内存地址，可显示相应地址的内容。该窗口的显示值可以以各种形式显示，如十进制、十六进制、字符型等。改变显示方式的方法是单击鼠标右键，在弹出的快捷菜单中选择。除了显示，还可以修改内存中的值。

查看和调用栈窗口（见图 2-18）可以帮助我们查看当前调用树的情况，以及通过这个窗口查看和修改一些变量的值。鼠标停留在某个变量的时候单击右键，在弹出的浮动菜单中选择 Add ***to Watch window，Local 窗口显示当前一些局部变量的值，变量值的显示方式可以在十六进制和十进制之间切换。方式是在查看窗口点右键，在某个变量的 Value 栏用鼠标单击然后按 F2（鼠标连续单击两次），即可修改该值。

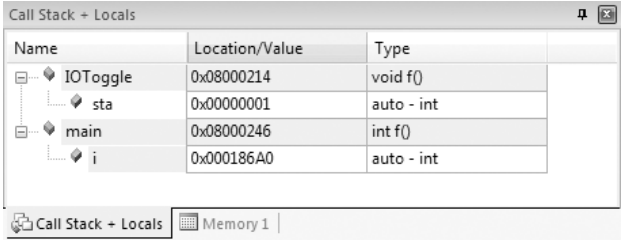


图 2-18　查看和调用堆栈窗口

2.2　仿　真　器

2.2.1　ST-Link

ST-Link 是 ST 公司针对 ST 公司产品的仿真器，目前被多数 ARM 开发环境支持，例如 MDK、IAR 等主流开发环境。其中，ST 公司的 Discovery 开发板提供 ST-Link/V2 嵌入式调试器接口板，但该调试器仅提供 SWD 接口，即 CN4 插针。该板上的 SWD 接口与常用的 Jtag 形式接口针数、排序也一致。图 2-19 是常用的 20 针的 Jtag 的接口与 Discovery 中的 CN4 连线图。

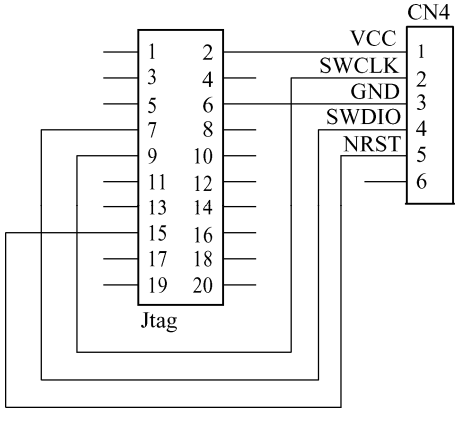


图 2-19　Discovery 的 ST-Link 转成 20 针接口

Jtag 是常用的间距 2.54mm 的 20 针插座，CN4 是 Discovery 板上的 CN4。作为仿真器时需要将 Discovery 开发板 CN5 的跳线都拔掉。

2.2.2　J-Link 与 U-Link2

J-Link 是 SEGGER 公司为支持仿真 ARM 内核芯片推出的 JTAG 仿真器，除了支持 MDK 编译环境，还可用于多种编译环境，在国内被大量使用。U-Link2 接口仿真器，是 Keil 公司的一款多功能 ARM 调试工具，仅针对 Keil 公司的 MDK 产品。J-Link/U-Link2 与 ST-Link 区别是 J-Link 与 U-Link2 目前支持所有 ARM 芯片的开发。

该两款仿真器用于软件开发时间，均需要在图 2-10 中选择相应的仿真器，并在图 2-11 中配置成 SWD 设置。有关 Flash 编程算法配置与 ST-Link 方式相同。

2.3　WinMerge

WinMerge 是一款运行于 Windows 系统下的免费开源的文件比较/合并工具，适合比较多个文档内容甚至是文件夹与文件夹之间的文件差异。软件可以从 http://winmerge.org/下载。

下面先说明为何在书中介绍一款与 STM32F0 无任何关联的软件，也可以说 WinMerge 是用来解决什么实际问题的。

STM 公司产品线做得比较细腻，更新比较快。STM32F0 家族又区分了很多型号，有不同也有相同之处。固件库伴随新的类型芯片推出而不断更新。而且除了固件库，另外还有 Discovery 例程、STM32Cube 例程。WinMerge 可以帮你了解这些软件之间的差异。而且 WinMerge 可集成到 TortoiseSVN 软件中，便于版本管理以及文件合并。

WinMerge 功能如下。

（1）支持文件（Windows、UNIX 和 Mac 文本文件格式）及文件夹的比较；对文本文档的可视化编辑以及合并。

（2）具有灵活的编辑器，支持语法高亮，显示行号和自动换行，差异窗口显示，在文件比较中检测移动过的段落。

（3）基于正则表达式的文件过滤器，允许排除和包含项目，支持比较文件夹内所有子文件夹。

（4）以树状形式显示文件夹比较结果。

（5） 支持插件，可添加 7zip 插件以及 xdocdiff 插件，使得 WinMerge 支持压缩文件比较以及 Word、Excel、PowerPoint、PDF 文件的比较。

以比较两个 STM32F0 的固件库 STM32F0xx_StdPeriph_Lib_V1.3.1 与 STM32F0xx_StdPeriph_Lib_V1.0.0 为例说明 WinMerge 用法。从 Winmerge.org 网站下载安装后，运行 WinMerge 软件。单击“文件”菜单下的“打开”命令，弹出图 2-20 所示对话框。单击“浏览”按钮，分别选择相应 STM32F0xx_StdPeriph_Lib_V1.3.1 与 STM32F0xx_StdPeriph_Lib_V1.0.0 目录。如果需比较某些特定文件，可通过配置过滤器。单击“确定”按钮开始比较。图 2-21 为目录比较结果。

图 2-21 是 WinMerge 目录比较结果。选中 stm32f0xx.h 文件，可查看两文件差异，

如图 2-22 所示。在该文件中可看到由于 STM32F0 家族芯片不断增加，有关外设宏定义在不断增加。在图 2-22 中，可将两个文件差异进行合并，减少程序变更工作量。

图 2-20　WinMerge 选择文件夹

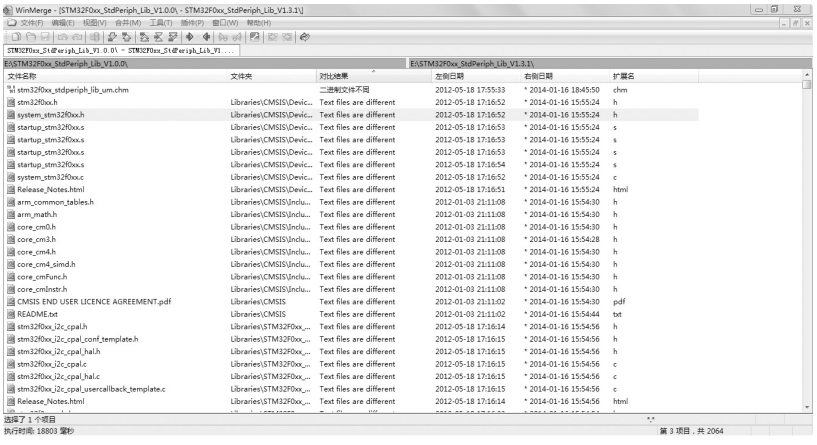


图 2-21　目录比较结果

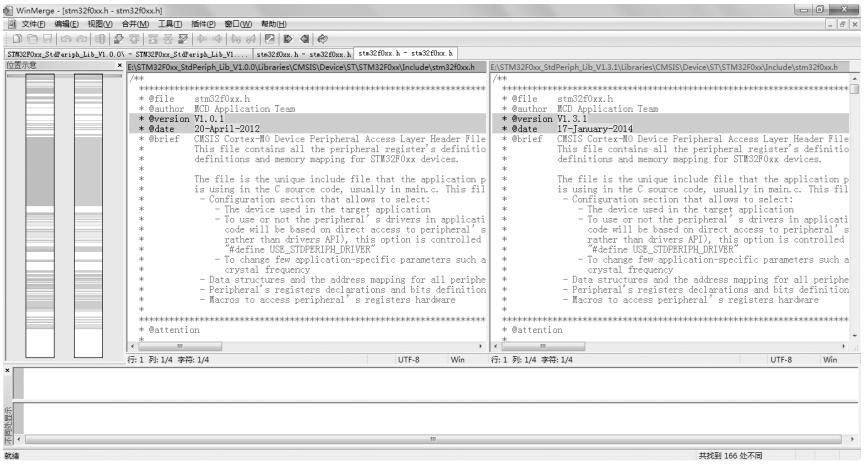


图 2-22　文件比较结果

2.4　小　　结

本章分别从编译已有工程以及新建工程两个角度说明了 MDK 用法，方便大量 C51 开发者熟悉 STM32F0 的开发环境。并介绍了 WinMerge 的使用，说明了为何要介绍该软件以及如何将其应用到 STM32F0 的开发过程中。

第 3 章

硬 件 基 础

本章是关于 STM32F0x 家族功能的总体介绍，主要包括系统存储器、电源控制、时钟以及对应的固件库函数内容，是进行硬件设计的基础。3.6 节以 STM32F0x 家族中最便宜的芯片 STM32F030F4P6 为例讲解 STM32F0x 硬件设计。

3.1 STM32F0 产品特征

STM32F0 家族主要包含 STM32F0x8、STM32F0x2、STM32F0x1、STM32F030 几个子系列，可用于手持设备、A/V 接收机、数字电视、PC 外设、电动自行车、打印机、扫描仪、报警系统、视频对讲、空调系统、PLC、工业控制设备。STM32F0 家族具有一些共同特点，均是基于 Cortex-M0 内核，可运行在 48MHz，带有 PDR/POR 等特征。各个子系列之间也有一些区别，可根据各自外设特点用于不同应用场合，如图 3-1 所示。例如，STM32F0x8 系列使用 1.8V 电压，适合用于消费电子应用，如智能电话、配件、媒体设备。

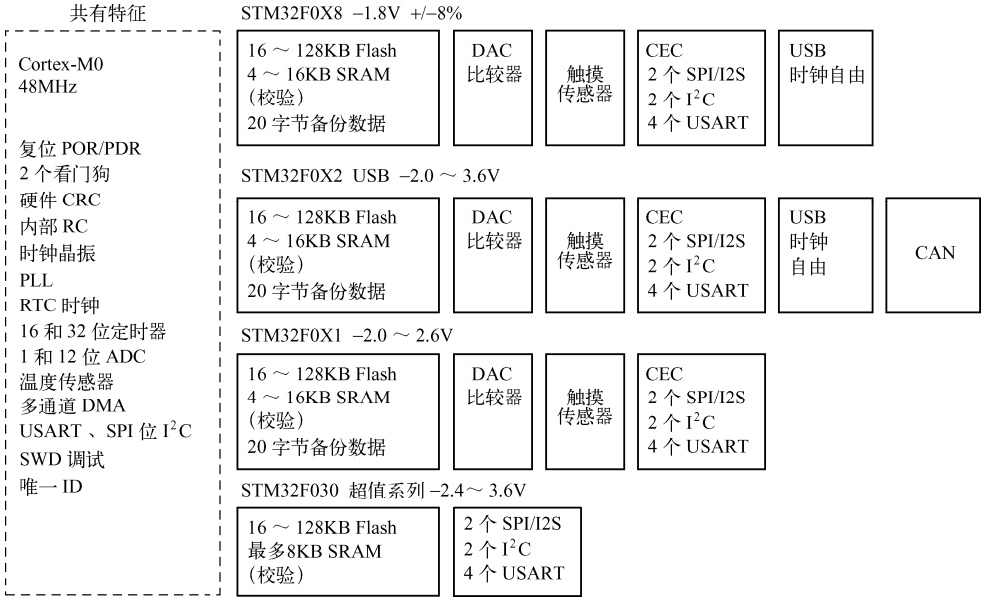


图 3-1　STM32F0 性能对比

3.2　系统及存储器概述

3.2.1　系统构架

图 3-2 是 STM32F0 的系统结构图，系统主要由 Cortex-M0 内核以及高性能总线（AHB）、通用 DMA 系统、内部 SRAM、内部闪存存储器、AHB 到 APB 的桥、专门用于连接 GPIO 口的 AHB2 构成。内部是一个多层 AHB 互联的系统总线。

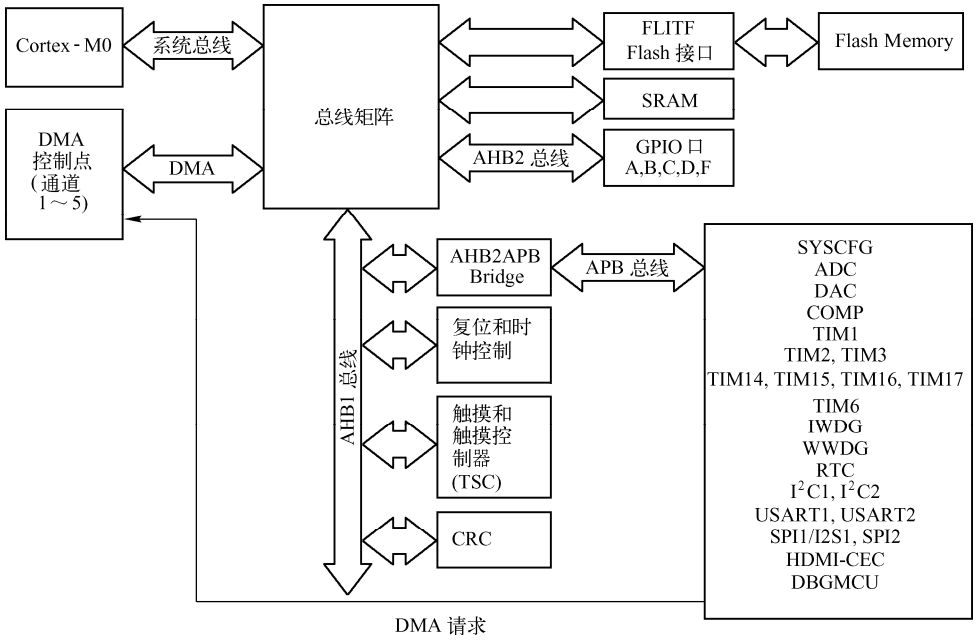


图 3-2　系统架构图

系统总线连接了 Cortex-M0 内核的系统总线与总线矩阵（BusMatrix），总线矩阵用来协调内核和 DMA 间的总线访问控制。DMA 总线将 DMA 的 AHB 主控接口与总线矩阵相连，总线矩阵协调 CPU 和 DMA 到 SRAM、闪存和外设的访问控制。总线矩阵管理着内核系统总线与 DMA 总线的访问仲裁，总线矩阵由 2 个主模块总线（CPU AHB、系统总线、DMA 总线）及 4 个从模块总线（FLIFT、SRAM、AHB2GPIO 和 AHB2APB 桥）组成。AHB 外设通过总线矩阵与系统总线相连，允许 DMA 访问。AHB 到 APB 桥（AHB2APB Bridge）在 AHB 与 APB 总线间提供同步连接。在每次复位之后，所有的外设时钟都关闭（ 除了 SRAM 及 FLIFT 外）。在用一个外设前，必须打开相应的 RCC_AHBENR、RCC_APB2ENR 或 RCC_APB1ENR 寄存器中时钟使能位。

注：当对 APB 寄存器进行 8 位或者 16 位访问时，该访问会被自动转换成 32 位的访问：桥会自动将 16 位或者 8 位的数据扩展以配合 32 位的宽度。

3.2.2 存储器组织

程序存储器、数据存储器、寄存器及 I/O 口统一编址，其线性地址空间达到 4G。数据字节以小端格式存放在存储器中，一个字里的最低地址字节被认为是该字的最低有效字节，而最高地址字节是最高有效字节。寻址空间分成 8 块，每块 512MB。其他所有没有分配给片上存储器和外设的存储器空间都是保留的地址空间。存储器映像和外设寄存器编址可在 STM32F0x 的参考手册中查阅。

3.2.3 启动配置

在 STM32F0xx 中，可通过 BOOT0 及 BOOT1 脚的配置选择三种不同的启动模式，见表 3-1。

表 3-1 启动模式

启动模式选择		启动模式	说　明
BOOT1	BOOT0		
x	0	主闪存存储器	主闪存存储器选为启动区域
1	1	系统存储器	系统存储器选为启动区域
0	1	内置SRAM	内置SRAM选为启动区域

器件复位后，在 SYSCLK 的第 4 个上升沿锁存 BOOT0 和 BOOT1 的引脚值，用户可通过设置 BOOT1 和 BOOT0 来选择启动模式。从待机模式唤醒时，CPU 会重新采样 BOOT0 及 BOOT1 的引脚值，因此在有待机应用的场合需要保持启动模式的设置。在启动延迟之后，CPU 从地址 0x0000 0000 获取堆栈顶的地址，并从启动存储器的 0x0000 0004 指示的地址开始执行代码。

根据选定的启动模式、主闪存存储器、系统存储器或 SRAM 按照以下的说明访问。

- 从主闪存存储器启动：主闪存存储器被映射到启动存储空间（0x0000 0000），但仍然能从原有的地址空间（0x800 0000）访问，即闪存存储器的内容可从 0x0000 0000 或 0x800 0000 两个地址开始访问。
- 从系统存储器启动：系统存储器被映射到启动空间（0x0000 0000），但仍然能够在它原有的地址空间（0x1FFF EC00）访问。
- 从内置的 SRAM 启动：SRAM 映射到启动空间（0x0000 0000），但其仍然能够在它原有的地址空间（0x2000 0000）访问。

一旦 BOOT 引脚被选用，应用程序代码可通过 SYSCFG 配置寄存器 1（SYSCFG_CFGR1）修改存储器访问方式，Cortex-M0 与 Cortex-M3、Cortex-M4 不同，它不支持向量表重定位。可采用如下方式重定位向量表到内部 SRAM。

- 从 Flash 复制向量表到基址为 0x2000 0000 的 SRAM。
- 使用 SYSCFG 配置寄存器 1，重映射 SRAM 到地址 0x0000 0000。
- 一旦发生中断，Cortex-M0 处理器将存取 SRAM 中重定位的中断句柄，而后跳转到 Flash 中的中断句柄。

3.3　电源控制（PWR）

3.3.1　电源

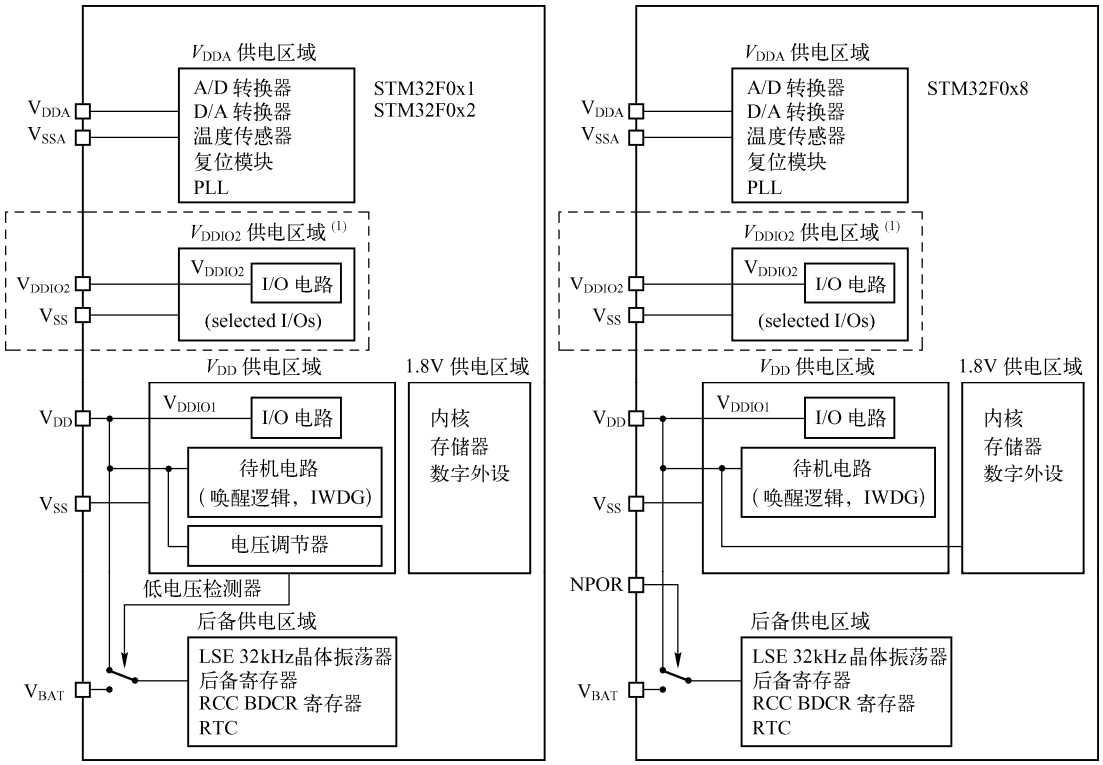
STM32F0x1/F0x2 子系列内嵌了提供内部 1.8V 的电压调节器，如图 3-3 所示。

（1）STM32F0x1/F0x2 需要 2.0～3.6V 的工作电压（V_{DD}）和 2.0～3.6V 的模拟电源电压。

（2）STM32F0x8 需要 1.8V±8%工作电源电压（V_{DD}）和一个 1.65～3.6V 模拟电源电压。

当主 V_{DD} 电源掉电时，实时时钟（RTC）和后备域寄存器可由可选的电池供电或由其他形式的电源供电。

STM32F03xxx 子系列内嵌了 1.8V 电压调节器。外部需要提供 2.4～3.6V 的工作电压（V_{DD} 和 V_{DDA}）。



注：（1）仅用于 STM32F07x。

图 3-3　电源供电框图

V_{DD} 主要为 GPIO 和内部调压器供电，V_{DDA} 主要是 ADC、DAC 和复位模块、内部谐振器、PLL、比较器、温度传感器供电，V_{DDA} 超过 2.4V，ADC 和 DAC 才能工作；当 V_{DD} 不在时给备份域供电。V_{DD} 和 V_{DDA} 可以来自不同的电源，V_{SS} 和 V_{SSA} 必须接地。电压调节器在运

行模式和停止模式下分别以全功耗模式、低功耗模式方式提供 1.8V 电源。

STM32F07x 系列采用了专门的 V_{DDIO2} 供电，范围是 1.65～3.6V，V_{DDIO2} 电压完全独立于 V_{DD} 与 V_{DDA}。V_{DDIO2} 电源被内部参考电压（V_{REFINT}）监视与比较。当低于阈值时间，该供电域的 I/O 引脚被禁用。EXTI 31 反映了比较结果并可用作中断。

STM32F0 的模拟地和数字地与其他各类芯片（例如，NXP 的 LPC22 系列、LPC17xx 系列以及 TI 的 2812 芯片）一样需要隔离，电源引脚是 V_{DDA}，模拟地是 V_{SSA}。这样做的好处是可提高转换精度，过滤和屏蔽来自电路板的噪声。

V_{DDA} 供电/参考电压可以大于等于 V_{DD} 电压。当器件用单电源供电方式，则 V_{DDA} 可连接到 V_{DD}，通过外部滤波电路以确保无噪声的 V_{DDA}/参考电压。当 V_{DDA} 与 V_{DD} 不同时，V_{DDA} 必须高于或等于 V_{DD} 的供电电压。在通电或断电的过程中，为了也确保这一条件的成立可在 V_{DD} 和 V_{DDA} 之间串接一个肖特基二极管。

在 STM32F07X 设备中，部分 I/O 的供电是独立的，通过 V_{DDIO2} 引脚供电，电压范围是 1.65～3.6V。电压水平与 V_{DD}、V_{DDA} 不同。V_{DDIO2} 电源被内部参考电压（V_{REFINT}）监视与比较，低于门限值，对应 IO 将被禁用，比较器输出连到 EXTI 线 31 上，可产生中断。

当关闭 V_{DD} 时间，V_{BAT} 可连接到电池或者其他电源上，由 V_{BAT} 引脚为 RTC 模块、LSE 振荡器和 PC13～PC15 供电。该切换过程由复位模块中的掉电复位（PDR）或者外部 NPOR 信号（STM32F0x8）控制。

当使用电池方式供电，推荐在 VBAT 引脚通过一个 100nF 的陶瓷电容与 V_{DD} 相连。当 RTC 由 V_{DD} 供电时，可以使用下述功能。

（3）PC13、PC14 和 PC15 可用于 GPIO，但引脚电流与速率受限。

（4）PC13、PC14 和 PC15 可通过 RTC 或者 LSE 配置。

3.3.2 电源管理器

电源管理主要涉及上电复位（POR）和掉电复位（PDR），如图 3-4 所示。电源管理器的内部有一个完整的上电复位（POR）和掉电复位（PDR）电路，当供电电压达到 2V 时系统即能正常工作。

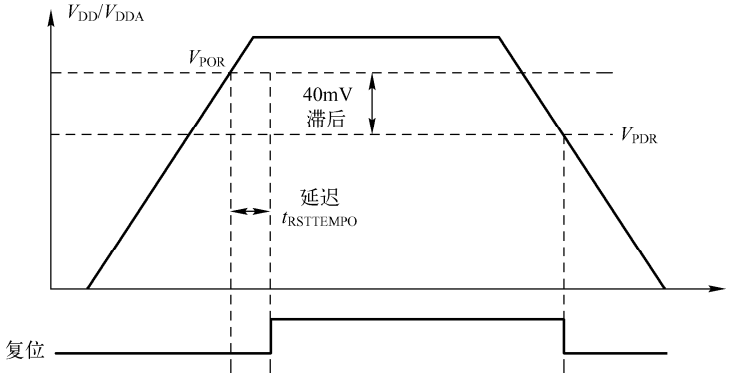


图 3-4　复位/掉电电源复位波形

当 V_{DD}/V_{DDA} 低于指定的门限电压 V_{POR}/V_{PDR} 时，系统保持为复位状态，而无须外部复位

电路。

- POR 仅检测 V_{DD} 电源，在启动阶段，V_{DDA} 需上电并大于或等于 V_{DD}。
- PDR 检测 V_{DD} 和 V_{DDA} 电源电压，但 V_{DDA} 电源是可被禁用（通过 $V_{DDA\_MONITOR}$ 选项位设置）以降低功耗。

门限值 V_{POR}/V_{PDR} 的设置以及状态信息在电源控制/状态寄存器（PWR_CSR）与控制寄存器（PWR_CR）中。

STM32F0x8 的 PB2（或者小封装的 PB1）的 I/O 功能不可用，被用作 NPOR，作为电源复位功能。NPOR 引脚内部上拉连接到 V_{DDA}。

3.3.3　低功耗模式

为降低功耗，CPU 可工作在低功耗模式下，有三种低功耗模式。

- 睡眠模式（Sleep mode）（CPU 时钟关闭，所有外设，包括内核外设，如 NVIC、SysTick 等仍在运行）
- 停止模式（Stop mode）（所有时钟都停止）
- 待机模式（Standby mode）（1.8V 供电域掉电）

即使在正常运行模式下，可通过降低系统时钟（SYSCLK、HCLK、PCLK）频率（SYSCLK、HCLK、PCLK）以及关闭未用外设的时钟方式减少功耗。表 3-2 是各种低功耗模式的情况。

表 3-2　低功耗模式概述

模　式	进　入	唤　醒	对1.8V区域时钟的影响	对V_{DD}区域时钟的影响	电压调节器
睡 眠	WFI	任意中断	CPU时钟关闭，对其他时钟及模拟时钟无影响	无	开
	WFE	唤醒事件			
停机	PDDS和LPDS位 + SLEEPDEEP位 + WFI或WFE	任意外部中断（在EXTI寄存器中设置）指定通信口接收事件（CEC、USART、I^2C）	所有1.8V区域时钟关闭	HSI 和 HSE 振荡器关闭	在低功耗模式可进行开/关设置（取决于电源控制寄存器PWR_CR）
待机	PDDS 位 +SLEEPDEEP 位 + WFI或WFE	WKUP 引脚上升沿、RTC 报警、NRST 的外部复位、IWDG 复位			关

RTC 可以在不需要依赖外部中断的情况下唤醒低功耗模式下的微控制器（自动唤醒模式)。RTC 提供一个可编程的时间基数，用于周期性从停止或待机模式下唤醒。通过对备份区域控制寄存器（RCC_BDCR）的 RTCSEL[1:0]位的编程，三个 RTC 时钟源中的两个时钟源可以选作实现此功能。

- 低功耗 32.768kHz 外部晶振（LSE）：该时钟源提供了一个低功耗且精确的时间基准（在典型情形下消耗小于 1μA）。
- 低功耗内部 RC 振荡器（LSIRC）：使用该时钟源，节省了一个 32.768kHz 晶振的成本，但是 RC 振荡器将少许增加电源消耗。

为了用 RTC 闹钟事件将系统从停止模式下唤醒，必须进行如下操作。

- 配置外部中断线 17 为上升沿触发。
- 配置 RTC 使其可产生 RTC 闹钟事件。

如果要从待机模式中唤醒，不必配置扩展中断线 17。

3.3.4 PWR 固件库

固件库中的 stm32f0xx_pwr.c 文件包含了有关电源控制外设（PWR）的一些常量、宏定义和相关的操作函数。该文件主要包含备份域访问、PVD 的配置、唤醒引脚配置、低电源模式配置和标志位管理。表 3-3 是对应函数说明。

表 3-3　PWR 的固件库函数

函　　数	功 能 说 明
void　PWR_BackupAccessCmd (FunctionalState NewState)	使能或禁用访问后备域寄存器
void　PWR_ClearFlag (uint32_t PWR_FLAG)	清除PWR的挂起标志
void　PWR_DeInit (void)	释放PWR外设，PWR外设寄存器恢复到默认复位状态
void　PWR_EnterSleepMode (uint8_t PWR_SLEEPEntry)	进入睡眠模式
void　PWR_EnterSTANDBYMode (void)	进入待机状态
void　PWR_EnterSTOPMode (uint32_t PWR_Regulator, uint8_t PWR_STOPEntry)	进入停止模式
FlagStatus　PWR_GetFlagStatus (uint32_t PWR_FLAG)	检查指定PWR表示是否设置

3.4　复位和时钟控制（RCC）

3.4.1　复位

STM32F0x 有三种复位：电源复位、系统复位和备份域复位，如图 3-5 所示。

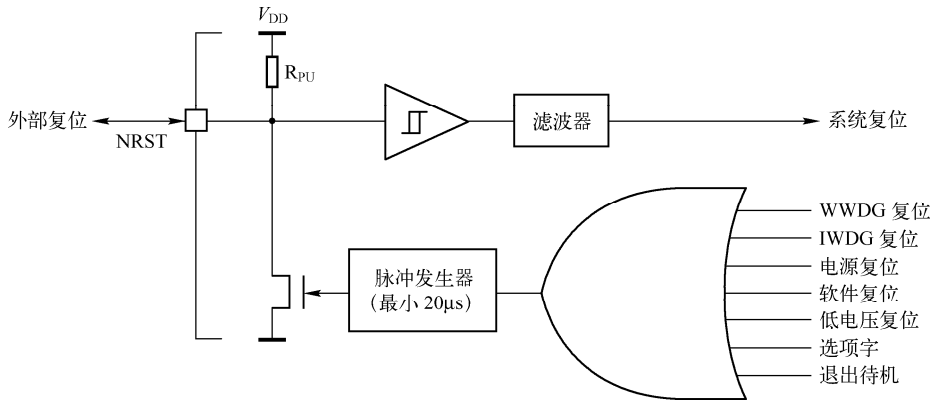


图 3-5　复位电路简化图

上电/掉电复位（POR/PDR 复位）以及从待机模式中返回均会产生电源复位。电源复位将复位除了备份域外的所有寄存器。由于 STM32F0X8 的 POR/PDR 不起作用以及待机模式

不可用，由 NPOR 引脚提供电源复位。

系统复位将复位除时钟控制寄存器 CSR 中的复位标志和备份域中的寄存器以外的所有寄存器，将它们复位成复位值。NRST 引脚上的低电平（外部复位）、窗口看门狗事件（WWDG 复位）、软件复位（SW 复位）、低功耗管理复位、选项字节装载器复位、电源复位均可产生一个系统复位。

可通过查看 RCC_CSR 控制状态寄存器中的复位状态标志位识别复位事件来源。图 3-5 中的复位源将最终反映到 NRST 引脚上，并在一定的延时时段内保持低电平。复位入口地址被固定在 0x0000 0004 处。内部的复位信号会在 NRST 引脚上输出，脉冲发生器保证每一个（外部或内部）复位源都能有至少 20μs 的脉冲延时；当 NRST 引脚被拉低产生外部复位时，它将产生复位脉冲。

以下两种情况下产生低功耗管理复位。

- 在进入待机模式时产生低功耗管理复位：通过将用户选择字节中的 nRST_STDBY 位置 1 将使能该复位。这时，即使执行了进入待机模式的过程，系统将被复位而不是进入待机模式。
- 在进入停止模式时产生低功耗管理复位：通过将用户选择字节中的 nRST_STOP 位置 1 将使能该复位。这时，即使执行了进入停机模式的过程，系统将被复位而不是进入停机模式。

备份域拥有两个专门的复位，它们只影响备份域。当以下事件中之一发生时，产生备份域复位。

- 软件复位，由备份域控制寄存器（RCC_BDCR）的 BDRST 位触发。
- 如果 VBAT 未连接，VDD 掉电。

如果出现 RTC 入侵检测时间或者读保护从 1 级改变到 0 级也会引起备份域寄存器的复位。

通过将 Cortex-M0 应用中断和复位控制寄存器中的 SYSRESETREQ 位置 1，可实现软件复位。

3.4.2　时钟

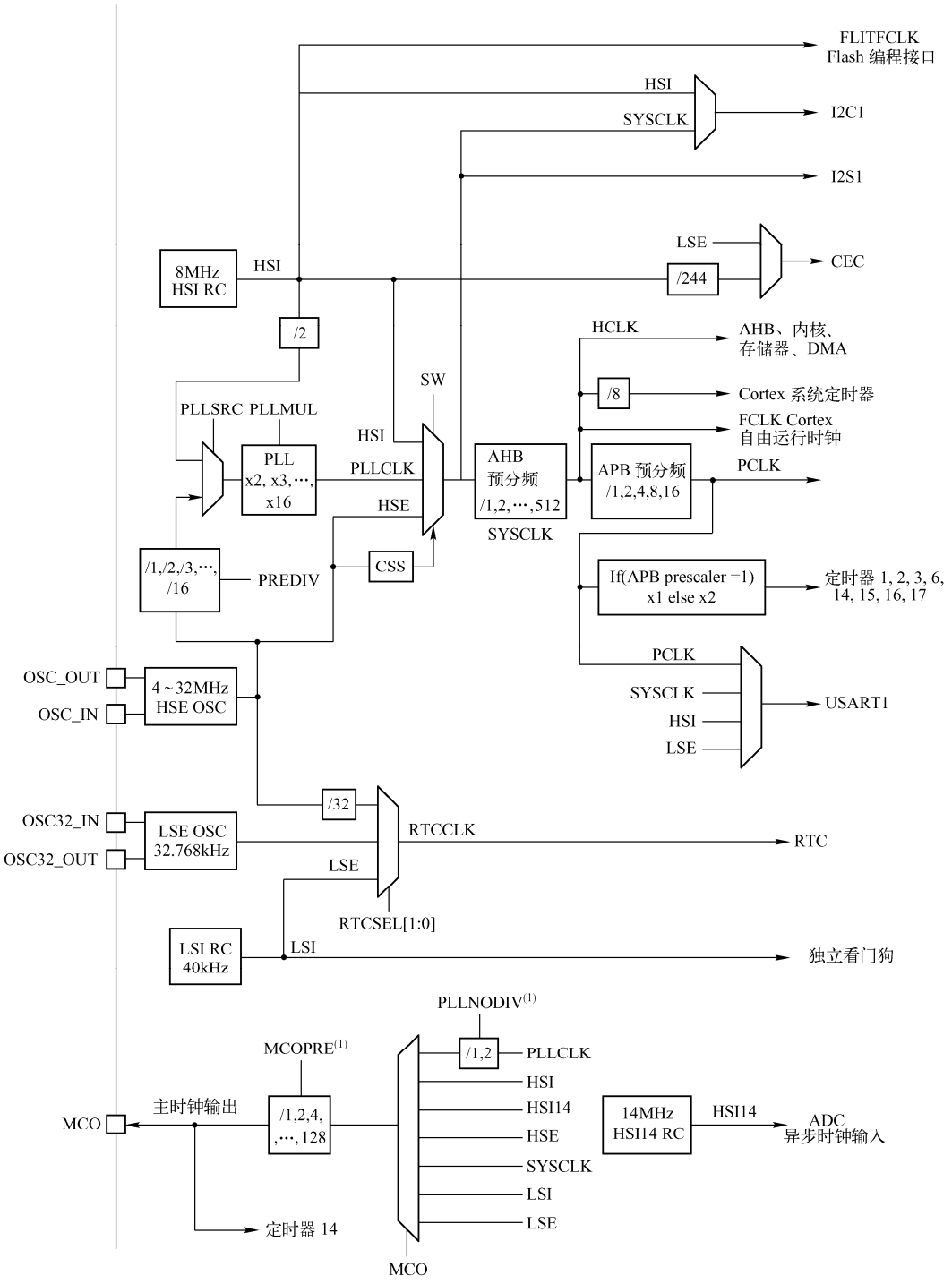
STM32F0x 的时钟源分为两类，一类用于驱动系统时钟，一类用于驱动特殊功能外设源。图 3-6 是 STM32F0x 的时钟树。有四种不同的时钟源可被用来驱动系统时钟：内部高速（HSI）8MHz RC 振荡器时钟、外部高速（HSE）振荡器时钟、PLL 时钟、HSI48 48MHz RC 振荡器时钟（仅用于 STM32F07x 列）。

下面的三种时钟驱动非系统时钟的时钟源。

- 40kHz 低速内部 RC 振荡器（LSI RC）：用于驱动独立看门狗和用于自动从停机或待机模式下唤醒的 RTC 时钟。
- 低速用于驱动实时时钟（RTCCLK）的 32.768kHz 低速的外部晶体（LSE 晶体）。
- 专门用于 ADC 的 14MHz 高速内部 RC 振荡器（HSI14）：每种时钟源都可以单独地打开或关断，当它们不用时，可以关断它们来降低功耗。

有多个分频器可用于配置 AHB 和 APB 时钟域。AHB 和 APB 域的最大时钟频率为 48MHz。Cortex 系统定时器由 AHB 时钟驱动，可由 AHB/8 或 AHB 时钟频率直接驱动（通

过 Cortex SysTick 配置位来配置）。



注：1 在 STM32F05x 中不可用。FCLK 是 Cortex-M0 的自由运行时钟。

图 3-6　时钟树

除了下面列出的，其余的外设时钟由其所在的总线时钟（HCLK 或 PCLK）驱动。

- 闪存编程接口时钟（FLITFCLK）总是由 HSI 时钟驱动。
- 选项字节装载器时钟也由 HIS 时钟驱动。
- ADC 的时钟源（软件选择）：专门的 HSI14 时钟、APB（PCLK 时钟除 2 或除 4）。
- USART1 时钟源（软件选择）：系统时钟、HSI 时钟、LSE 时钟、APB 时钟（PCLK）。
- I2C1 的时钟源（软件选择）：系统时钟、HSI 时钟。
- CEC 时钟来自于 HSI/244 或者 LSE。
- I2S1 时钟为系统时钟。
- RTC 时钟来自于 LSE、LSI 或 HSE/32。
- IWWDG 时钟永远来自 LSI。

RCC 通过 AHB 时钟（HCLK）8 分频后作为 Cortex 系统定时器（SysTick）的外部时钟。通过对 SysTick 控制与状态寄存器的设置，可选择上述时钟或 Cortex（HCLK）时钟作为 SysTick 时钟。

（1）HSE 时钟

高速外部时钟信号（HSE）由 HSE 外部晶体/陶瓷谐振器或者 HSE 用户外部时钟产生。为了减少时钟输出的失真和缩短启动稳定时间，晶体/陶瓷谐振器和负载电容必须尽可能地靠近振荡器引脚。负载电容值必须根据所选择的振荡器来调整。

4～32MHz 外部振荡器可为系统提供非常精确的主时钟。在时钟控制寄存器（RCC_CR）中的 HSERDY 位用来指示高速外部振荡器是否稳定。在启动时，直到这一位被硬件置 1，该时钟才可使用。如果在时钟中断寄存器（RCC_CIR）中允许产生中断，将会产生相应中断。

HSE 晶体可以通过设置时钟控制寄存器（RCC_CR）中的 HSEON 位被启动和关闭。在外部时钟源（HSE 旁路）模式里，必须提供外部时钟。它的频率最高可达 32MHz。用户可通过设置在时钟控制寄存器（RCC_CR）中的 HSEBYP 和 HSEON 位来选择这一模式。占空比为 40%～60%外部时钟信号（方波、正弦波或三角波）必须连到 SOC_IN 引脚，同时保证 OSC_OUT 不作为 IO 口使用。

（2）HIS 时钟

HSI 时钟信号由内部 8MHz 的 RC 振荡器产生，可直接作为系统时钟或在 2 分频后作为 PLL 输入。HSI 振荡器能够在不需要任何外部器件的条件下提供系统时钟。它的启动时间比 HSE 晶体振荡器短。然而，即使在校准之后它的时钟频率精度仍较差（相比于晶体振荡器或陶瓷谐振器）。

制造工艺决定了不同芯片的 RC 振荡器频率会不同，这就是为什么每个芯片的 HSI 时钟频率在出厂前已经被 ST 校准到 1%（25℃）的原因。系统复位时，工厂校准值被装载到时钟控制寄存器的 HSICAL[7:0]位中（HSICAL 在时钟控制寄存器 RCC_CR 中）。

如果用户的应用基于不同的电压或环境温度，这将会影响 RC 振荡器的精度。可以通过时钟控制寄存器（RCC_CR）里的 HSITRIM[4:0]位来调整 HSI 频率。

时钟控制寄存器（RCC_CR）中的 HSIRDY 位用来指示 HSI RC 振荡器是否稳定。在时钟启动过程中，直到这一位被硬件置 1，HSI RC 输出时钟才可以被使用。

HSI RC 可由时钟控制寄存器（RCC_CR）中的 HSION 位来启动和关闭。如果 HSE 晶体振荡器失效，HSI 时钟会被作为备用时钟源。

（3）PLL

内部 PLL 可用 HSI 或 HSE 倍频得到，PLL 配置（选择输入时钟，倍频因子）须在使能 PLL 前配置好。一旦 PLL 使能，PLL 使用到的这些参数就不能被改变。

改变 PLL 配置过程如下：

① 设置 PLLON=0，禁用 PLL。

② 等待 PLLRDY 清 0。PLLRDY 清 0 时才表明 PLL 已经完全停止。

③ 改变 PLL 所需参数。

④ 设置 PLLON=1 重新使能 PLL。

当使能时钟中断寄存器（RCC_CIR）的相应位，当 PLL 时钟就绪时会产生一个中断。PLL 输出频率设置的范围是 16～48MHz。

（4）LSE 时钟

LSE 时钟是一个 32.768kHz 的低速外部晶体或陶瓷谐振器。它为实时时钟或者其他定时功能提供一个低功耗且精确的时钟源。晶体振荡器的开和关可用备份域控制寄存器（RCC_BDCR）中的 LSEON 位来控制。运行时改变 RCC_BDCR 中的 LSEDRV[1:0]值可改变晶振振荡器驱动能力。根据系统鲁棒性、短启动时钟及低功耗的要求进行 LSEDRV 选值。

备份域寄存器（RCC_BDCR）中的 LSERDY 位指示 LSE 晶体振荡是否稳定。在该位被硬件置为 1 之前，LSE 的时钟信号都不被使用。如果在时钟中断寄存器（RCC_CIR）里被允许，可产生中断申请。

（5）外部时钟源（LSE 旁路）

在这个模式里必须提供一个 32.768kHz 频率的外部时钟源。通过设置备份域控制寄存器（RCC_BDCR）里的 LSEBYP 和 LSEON 位来选择这个模式。具有 50%占空比的外部时钟信号（方波、正弦波或三角波）必须连到 OSC32_IN 引脚，同时保证 OSC32_OUT 引脚不作为 IO 口使用。

（6）LSI 时钟

LSI 时钟作为一个低功耗时钟源，它可以在停机和待机模式下保持运行，为独立看门狗和 RTC 提供时钟。LSI 时钟频率大约为 40kHz（在 30～60kHz 之间）。LSI 时钟振荡器可由时钟控制状态寄存器（RCC_CSR）中的 LSION 位来控制开或关。

在时钟控制/状态寄存器（RCC_CSR）里的 LSIRDY 位指示低速内部振荡器是否稳定。在这个位被硬件设置为 1 之前，LSI 时钟都不能被使用。如果时钟中断寄存器使能将产生 LSI 中断请求。

（7）系统时钟（SYSCLK）选择

HSI 振荡器、HSE 振荡器、PLL 均可用于驱动系统时钟（SYSCLK）。

系统复位后，HSI 振荡器被选为系统时钟。当时钟源被直接或通过 PLL 间接作为系统时钟时，它将不能被停止。时钟的切换只有在目标时钟源可用的情况下才能进行。假如系统选择了未准备好的时钟源作为当前系统时钟，那么只有在目标时钟源准备好之后才真正执行切换时钟源的操作。时钟控制寄存器（RCC_CR）指示当前系统时钟采用哪个时钟源作为系统时钟。

（8）时钟安全系统（CSS）

时钟安全系统可以通过软件被激活。一旦其被激活，时钟监测器将在 HSE 振荡器启动延迟后被使能，并在 HSE 时钟关闭后关闭。如果 HSE 时钟发生故障，HSE 振荡器被自动关闭，时钟失效事件将被送到高级定时器（TIM1 和 TIM8）的刹车输入，并产生时钟安全中断 CSSI，允许软件完成系统的补救处理。此 CSSI 中断连接到 Cortex-M0 的 NMI 中断（不可屏蔽中断）。

注意：一旦 CSS 被激活，并且 HSE 时钟出现故障，CSS 中断就产生，并且 NMI 也自动产生。NMI 将被不断执行，直到 CSS 中断挂起位被清除。因此，在 NMI 的处理程序中必须通过设置时钟中断寄存器（RCC_CIR）里的 CSSC 位来清除 CSS 中断。

如果 HSE 振荡器被直接或间接地作为系统时钟，时钟故障将导致系统时钟自动切换 HSI 振荡器，同时外部 HSE 振荡器被关闭。在时钟失效时，如 HSI 振荡器时是作为 PLL 的输入时钟，PLL 也将被关闭。

（9）ADC 时钟

ADC 时钟可从专门的 14MHz RC 振荡器（HSI14）或 PCLK/2（或/4）得到。当 ADC 时钟源于 PCLK 时，其 ADC 时钟为 PCLK 时钟的反相信号。14MHz 的 HSI RC 振荡器可以软件配置成由 ADC 接口控制的打开/关闭（自动关）模式或者常开模式。当 APB 时钟被选为内核时钟时，14MHz HSI RC 振荡器不能被 ADC 接口打开。

（10）RTC 时钟

通过设置备份域控制寄存器（RCC_BDCR）里的 RTCSEL[1:0]位，RTCCLK 时钟源可以由 HSE/32、LSE 或 LSI 时钟提供。除非备份域复位，此选择不能被改变。系统必须按 PCLK 的频率，须以快于或等于 RTCCLK 频率的方式配置才能正确操作 RTC。

LSE 时钟属于备份域的，但 HSE 和 LSI 时钟不属于备份域时钟，因此：

- ❑ 若 LSE 被选为 RTC 时钟，即使 VDD 掉电，RTC 仍会继续工作。
- ❑ 若 LSI 被选为 RTC 时钟，当 VDD 掉电时，RTC 处于不定的状态。
- ❑ 若 HSE/32 被选为 RTC 时钟，当 VDD 掉电或内部电压调压器（1.8V 域的供电切断）掉电时，RTC 处于不定的状态。

（11）看门狗时钟

如果独立看门狗已经由硬件选项或软件启动，LSI 振荡器将被强制在打开状态，并且不能被关闭。在 LSI 振荡器稳定后，时钟供应给 IWWDG。

（12）时钟输出

微控制器允许输出时钟信号到外部 MCO 引脚。对应 MCO 的 GPIO 口须配置为备用的功能模式。HSI14、SYSCLK、HIS、HSE、PLL/2 均可做 MCO 输出。MCO 时钟的选择由时钟配置寄存器（RCC_CFGR）MCO[2:0]位决定。

3.4.3　低功耗模式

APB 外设时钟和 DMA 时钟可用软件禁止。

睡眠模式停止 CPU 时钟。在 CPU 睡眠中，存储器接口时钟（Flash 和 RAM 接口）可被停止。当所有外设时钟被禁用，睡眠模式下的 AHB 连接 APB 桥时钟将被禁用。

CPU 进入停止模式时，停止 V18 域、PLL、HSI、HSI14 和 HSE 振荡器的时钟。

HDMI CEC、USART1 和 I2C1 即使在 MCU 进入停止模式下仍能打开 HIS 振荡器（假如 HIS 被选为这些外设的时钟）。

在睡眠模式并且 LSE 使能（LSEON）情况下，HDMI CEC、USART1 和 USART2（仅 STM32F07x）可由 LSE 振荡器驱动（如 LSE 被选用该外设时钟），但没有打开 LSE 振荡器的能力。

CPU 进入待机模式时，停止 V18 域、PLL、HSI、HSI14 和 HSE 振荡器的时钟。

当设置 DBGMCU_CR 寄存器中的 DBG_STOP 或 DBG_STANDBY 位，CPU 在相应的深度睡眠模式下也具有调试功能。

当系统由中断（停止模式）或复位（待机模式）唤醒后，HSI 振荡器被选为系统时钟（不管进入停止模式或待机模式前选用的是何种时钟）。

假如当前正在进行闪存编程，只有在闪存编程全部完成之后才会进入深度睡眠模式（深度睡眠延后）。若当前正在使用 APB 域，那么只有全部完成 APB 域的操作后才进入深度睡眠模式。

3.5 RCC 固件库

stm32f0xx_rcc.c 文件给出了 RCC 的固件库函数，主要提供了内部/外部时钟、PLL、CCS 和 MCO 配置，系统、AHB 和 APB 时钟配置，外设时钟配置，中断和标志管理，见表 3-4。

表 3-4 RCC 固件库函数

函 数	描 述
void RCC_ADCCLKConfig (uint32_t RCC_ADCCLK)	配置 ADC 时钟（ADCCLK）
void RCC_AdjustHSI14CalibrationValue (uint8_t HSI14CalibrationValue)	调整针对 ADC 的内部高速振荡器（HSI14）校准值
void RCC_AdjustHSICalibrationValue (uint8_t HSICalibrationValue)	调整内部高速振荡器（HIS）校准值
void RCC_AHBPeriphClockCmd (uint32_t RCC_AHBPeriph, FunctionalState NewState)	使能或者禁用 AHB 外设时钟
void RCC_AHBPeriphResetCmd (uint32_t RCC_AHBPeriph, FunctionalState NewState)	强制或释放 AHB 外设复位
void RCC_APB1PeriphClockCmd (uint32_t RCC_APB1Periph, FunctionalState NewState)	使能或者禁用低速 APB（APB1）外设时钟
void RCC_APB1PeriphResetCmd (uint32_t RCC_APB1Periph, FunctionalState NewState)	强制或者释放低速 APB（APB1）外设时钟
void RCC_APB2PeriphClockCmd (uint32_t RCC_APB2Periph, FunctionalState NewState)	使能或者禁用高速 APB（APB2）外设时钟
void RCC_APB2PeriphResetCmd (uint32_t RCC_APB2Periph, FunctionalState NewState)	强制或释放高速 APB（APB2）外设复位
void RCC_BackupResetCmd (FunctionalState NewState)	强制或释放后备域复位
void RCC_CECCLKConfig (uint32_t RCC_CECCLK)	配置 CEC 时钟（CECCLK）
void RCC_ClearFlag (void)	清除 RCC 复位标志
void RCC_ClearITPendingBit (uint8_t RCC_IT)	清除 RCC 的中断挂起标志
void RCC_ClockSecuritySystemCmd (FunctionalState NewState)	使能或禁用时钟安全系统

续表

函　数	描　述
void　RCC_DeInit (void)	复位 RCC 时钟配置为默认复位状态
void　RCC_GetClocksFreq (RCC_ClocksTypeDef *RCC_Clocks)	返回系统、AHB 和 APB 总线时钟的频率值
FlagStatus　RCC_GetFlagStatus (uint8_t RCC_FLAG)	检查指定的 RCC 表示是否设置
ITStatus　RCC_GetITStatus (uint8_t RCC_IT)	检查指定 RCC 中断是否发生
uint8_t　RCC_GetSYSCLKSource (void)	返回用于系统时钟的时钟源
void　RCC_HCLKConfig (uint32_t RCC_SYSCLK)	配置 AHB 时钟（HCLK）
void　RCC_HSEConfig (uint8_t RCC_HSE)	配置外部高速振荡器（HSE）
void　RCC_HSI14ADCRequestCmd (FunctionalState NewState)	使能或禁用 ADC 请求的内部高速振荡器
void　RCC_HSI14Cmd (FunctionalState NewState)	使能或禁用 ADC 对应的内部高速时钟
void　RCC_HSICmd (FunctionalState NewState)	使能或禁用内部高速时钟（HIS）
void　RCC_I2CCLKConfig (uint32_t RCC_I2CCLK)	配置 I2C1 时钟
void　RCC_ITConfig (uint8_t RCC_IT, FunctionalState NewState)	使能或禁用 RCC 中断
void　RCC_LSEConfig (uint32_t RCC_LSE)	配置外部低速振荡器
void　RCC_LSEDriveConfig (uint32_t RCC_LSEDrive)	配置外部低速时钟（LSE）
void　RCC_MCOConfig (uint8_t RCC_MCOSource)	时钟源输出到 MCO 引脚（PA8）
void　RCC_PCLKConfig (uint32_t RCC_HCLK)	配置 APB 时钟（PCLK）
void　RCC_PLLCmd (FunctionalState NewState)	使能或禁用 PLL
void　RCC_PLLConfig (uint32_t RCC_PLLSource, uint32_t RCC_PLLMul)	配置 PLL 时钟源和倍频
void　RCC_RTCCLKCmd (FunctionalState NewState)	使能或禁用 RTC 时钟
void　RCC_RTCCLKConfig (uint32_t RCC_RTCCLKSource)	配置 RTC 时钟（RTCCLK）
void　RCC_SYSCLKConfig (uint32_t RCC_SYSCLKSource)	配置系统时钟（SYSCLK）
void　RCC_USARTCLKConfig (uint32_t RCC_USARTCLK)	配置 USART1 时钟（USART1CLK）
ErrorStatus　RCC_WaitForHSEStartUp (void)	等待 HSE 启动

3.6　硬 件 设 计

STM32F0 家族中 STM32F030F4P6 价格便宜。如果仅仅作为学习用途，除了 CAN 与 USB 功能，采用 STM32F030F4P6 均可满足学习目的。即使设计产品，可根据产品需求以及 STM32F030F4P6 的外设资源情况决定是否采用。本书除了第 15 章，本节所给电路均可使用。

图 3-7 为 STM32F030F4P6 的核心电路，其中由 U3（SPX1117M3-3.3）输入 5V 电压，输出 3.3V 电压供给芯片。STM32F030F4P6 的 VDD 引脚与 VDDA 引脚，其中 VDDA 引脚电压与 VDD 引脚电压相同。由于 STM32F030F4P6 不含有 VSSA，即模拟地，所以

STM32F030F4P6 的 ADC 精度会受到影响。对于 STM32F0x 的其他芯片，推荐通过电感或0Ω电阻将 VDDA 与 VDD 隔离。

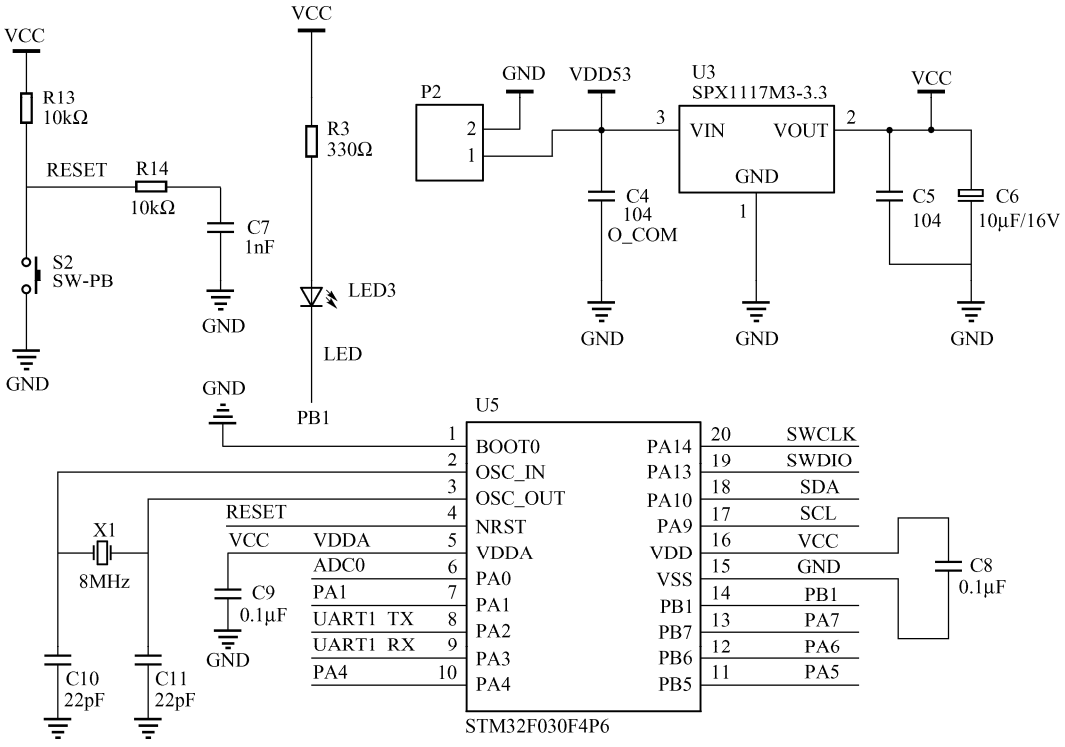


图 3-7　STM32F030F4P6 核心电路

除了 ST-Link 仿真器，目前 J-Link 和 U-Link 仿真器多数是 20 针的接口，所以继续采用 20 针端子形式。由于 STM32F0x 仅有 SWD 调试方式，所以图 3-8 中 P1 的 7 与 9 引脚连接到 SWDIO 与 SWCLK 上。

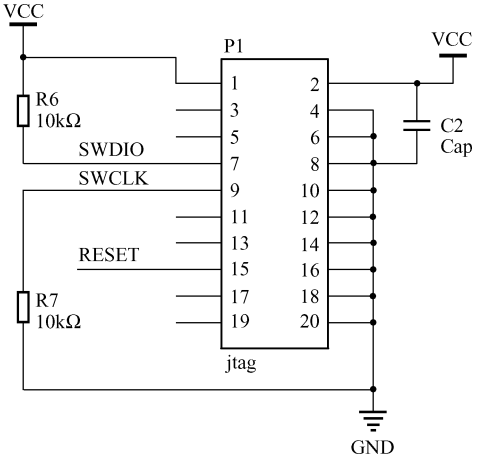


图 3-8　SWD 接口

图 3-9 为 STM32F0x 连接 24C16 的 I²C 接口的 EPROM，其中 R1 与 R2 为 SCL 与 SDA 的上拉电阻。

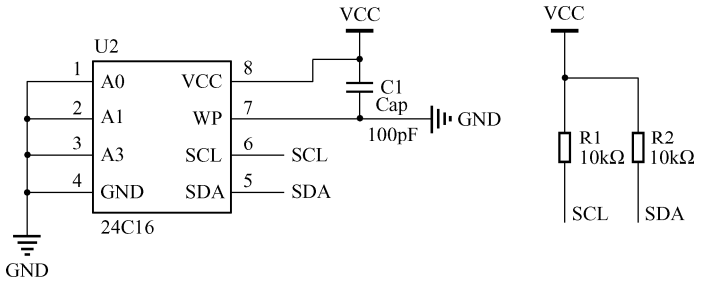


图 3-9　24C16 的 I²C 接口

图 3-10 是通过 PL2303HXD 将 USB 转换成 USART 信号。其中 R30 与 C15 用于静电保护。通过 USB 接口提供的 5V 电源可直接给图 3-7 中的 SPX1117M3-3.3 供电。PL2303HXD 无须提供外部晶振。

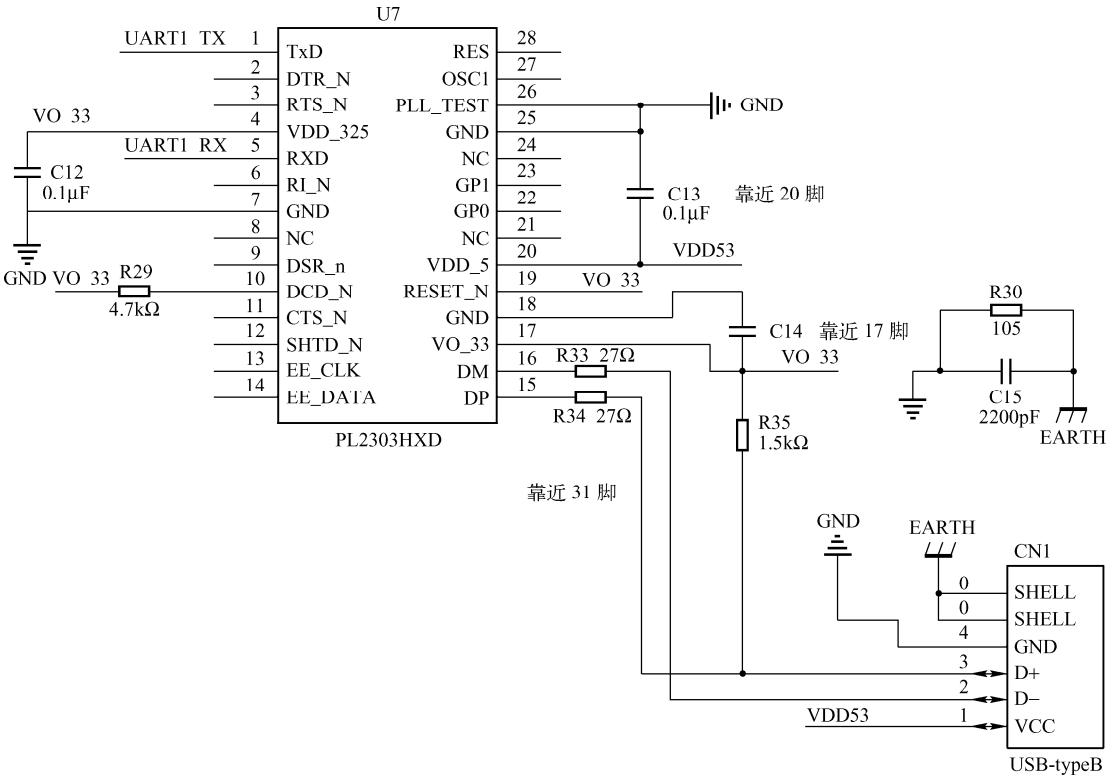


图 3-10　USB 转 USART 电平

注意：图 3-10 中的 USART 的 RX 与 TX 信号接法。图 3-10 中提供的接法为了将来扩展 USB 转 232/485 的接口，该方法充分利用了 STM32F0x 家族 RX 与 TX 可交换的特性。通用接法可参见第 7 章中的图 7-1。7.7 节说明了 USART_SWAPPinCmd(USART1,ENABLE)的用法。

3.7 小　　结

本章是关于 STM32F0x 家族的总体介绍，分别介绍了系统存储器、电源控制、时钟以及对应的固件库函数内容。3.6 节给出了 STM32F030F4P6 的硬件设计。作为 STM32F0x 家族中最便宜的芯片，使用其学习、设计产品都具有很大的优势，但也说明 STM32F030F4P6 芯片 ADC 方面的不足，在设计产品时根据产品需求加以抉择。

第4章

STM32F0 的固件库

由于 STM32F0 的开发几乎都是基于 C 语言，所以在 4.1 节首先讲解了 C 语言应用到嵌入式开发中的一些特点和限制，而后介绍了 ST 公司 STM32F0 固件库的组织结构和使用方法。

4.1 ARM 的 C 语言

4.1.1 嵌入式 C 语言的几个特殊之处

本节并不详细讲解 C 语言，而是给出一些关于 C 语言嵌入式系统开发的经验之谈。

在效率与代码简洁以及代码维护之间，倾向选择代码简洁与日后代码维护方便，而不是选择效率。例如，在嵌入式中经常会遇到将整数、浮点数转换成字符串的情况，用于界面显示。如果转换代码是在主循环中，笔者推荐使用 sprintf 函数，而不是使用除法和取模方式换算得到。当然对于中断里面的程序，不推荐该方式。在本节也会讲解如何将 Flash 代码复制到 RAM 中运行提高速度。在某些特殊情况慎用 C 语言的库函数，而是自己去实现。例如，整数的开平方运算，可结合牛顿迭代法根据需求进行求解，满足精度要求条件下进而提高运算速度。

unsigned 是代表无符号。下面的例子说明无符号的特殊之处，也许你不会犯如此低级错误，但作者在一个产品的数据计算中出现类似情况。

```
unsigned   int ui1 =1 ;
unsigned   int ui_n   = −1          ;
int i ;
i   =ui_n;//结果为 0xFFFFFFFF
if (ui1 >ui_n)
 {
     ;//实际结果是 ui1<ui_n
 }
 else
 {
     ; // 对于无符号数：−1 是大于 1 的。
```

```
}
```

C 语言中的指针就是指向内存某块区域的代名词，也是高级语言访问硬件的唯一方式。不仅仅可以指向寄存器，也可指向可执行代码。函数指针就是用来指向在内存中某块区域的执行代码。

在嵌入系统中，不可能采用 PC 键盘作为信息输入方式。图 4-1 是比较常见的嵌入式系统的用户输入方式，可能被用于输入参数信息，也可能被用于控制流程。在被用作控制程序流程时，常采用如下的 Swith 代码形式。

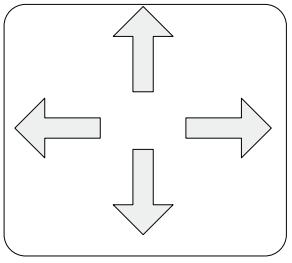


图 4-1　按键形式

```
switch (key)
    {
    case UP:
            UP_function();
    case Down:
            Down_function();
    case Left:
            Left_function();
    case Right:
            Right_function();
}
```

在《代码大全》一书中讲解了表驱动法（通俗叫法是查表法），即从表中查找信息而不是使用逻辑语句（if 或者 switch）。下面代码根据表驱动法并结合函数指针实现 Switch 功能。为了清晰地表示键值，首先定义一个枚举类型：

```
typedef   enum {
UP,
Down,
Left,
Right
}KeyValue
```

其次定义函数指针数组，并给数组赋值。

```
void (*fun[4])(void);
fun[0] = UP_function();
fun[1]= Down_function();
```

```
fun[0] = Left_function();
fun[1]= Right_function();
```

按键逻辑部分可以写成

```
（*fun[ key]）();//key 为按键值，对应 KeyValue 中的某一个值。
```

表格值不仅仅可以是函数指针，也可用于其他数据类型，用于表示查询到的数据结果。

4.1.2　寄存器访问方式总结

除了 ARM 的内核寄存器，ARM 的外设寄存器均采用内存映射方式，即访问寄存器是通过访问内存区域方式进行的，采用指针方式访问外设寄存器。例如如下形式：

```
#define SYST_RVR    (*((volatile unsigned long *) 0xE000E014))   /* SysTick Reload Value Register */
```

在其基础上，更多时候是将某一外设的相关寄存器放到一个结构体，例如 core_cm0.h 中的 SysTick 相关外设的寄存器定义如下。

（1）对 SysTick 相关寄存器进行结构体定义。

```
typedef struct
{
  __IO uint32_t CTRL;          /*!<偏移量: 0x000 (R/W)   SysTick 控制和状态寄存器   */
  __IO uint32_t LOAD;          /*!<  偏移量: 0x004 (R/W)   SysTick 重载值寄存器          */
  __IO uint32_t VAL;           /*!<  偏移量: 0x008 (R/W)   SysTick 当前计数值        */
  __I   uint32_t CALIB;        /*!<  偏移量: 0x00C (R/ )   SysTick 校准寄存器        */
} SysTick_Type;
```

（2）下面是 systick 外设的第一个寄存器地址位置，其余寄存器通过偏移量方式获得。

```
#define SCS_BASE              (0xE000E000UL)                    /*!<  系统控制基地址  */
#define SysTick_BASE          (SCS_BASE +    0x0010UL)          /*!< SysTick 基地址     */
```

（3）下面是进行指针的强制类型转换。

```
#define SysTick       ((SysTick_Type   *)    SysTick_BASE   )   /*!< SysTick 配置结构体      */
```

（4）访问 SysTick 控制和状态寄存器采用 SysTick-> CTRL 方式。

对于寄存器的内部位访问是通过位操作方式，当然也可以直接对寄存器赋值。但对寄存器直接赋值方式代码不清晰，代码的维护成本高，不推荐使用。通过位方式访问寄存器的常用操作如下。

（1）清零是使用与操作（&），并且对应位为 0。

例如，对单独的位操作：temp&=~(1<<2)；将第 2 位清零，其余位不影响。

（2）置位使用或操作。

例如：temp |=(1<<2)。

对某个寄存器的赋值，尽量不要直接赋某个具体数值的形式，应尽量使用 Reg1|=(1<<2)|(1<<6)，该方式表示将第 2 位与第 6 位置 1。

有时某个寄存器使用两个以上的比特位表示某一含义，可使用 Reg1|=(3<<15)。假设

Regs1 的 15 位、16 位、17 位共同表示某一含义，该方式表示将 15、16、17 位的数值置成 3。

（3）位读取判断。可使用 K&R 的《C 语言程序设计》一书中提供的读位判断函数。

```
/*
该函数是在“x”数据，获取从 p 开始的 n 个比特数据内容；
该函数利用了~0，与右移补 0 的问题。
*/
unsigned getbits(unsigned x ,int p, int n)
{ return (x>>(p+1-n)&~(~0<<n));}
```

4.1.3 struct 字节对齐

出于效率考虑，结构体（strcut）中的域默认方式是位于自然尺寸的边界：编译器会在域之间填充，保证它们是自然对齐的。strcut 的字节对齐是一个空间与效率之间选择问题。在通信类协议的场合推荐基于空间考虑，例如工业场合经常提及的 Modbus 协议。Modbus 协议中对离散量是以位方式表示的，对保持寄存器和输入寄存器规定是 16 位数据，但 ARM 是 32 位的。为了写程序方便，经常采用类似的方式。

```
typedef   union{
    int ComData[4]; //用于通信
  struct {
    char   inputreg1;
    int      inputreg2;
    char   inputreg3;
    short inputreg4;
  int   inputreg5;
}Data ;
} PData;
```

定义一个联合体的方式，其中 ComData 用于数据通信，Data 用于数据解释。假设通过通信后得到如下数据，其中 pTest 为 PData 定义的变量。但进行数据解释时，结果还是吃一惊的。图 4-2 是通过 MDK 中的数据排列，图 4-3 是 pTest 在内存中的摆放方式。

pTest	0x20000014 &pTest	union <untagged>
ComData	0x20000014 &pTest	int[4]
[0]	0x12345678	int
[1]	0x13579246	int
[2]	0x21436587	int
[3]	0x31427586	int
Data	0x20000014 &pTest	struct <untagged>
inputreg1	0x78 'x'	char
inputreg2	0x13579246	int
inputreg3	0x87 '?'	char
inputreg4	0x2143	short
inputreg5	0x31427586	int

图 4-2　MDK 中的数据排列

```
pTest.ComData[0]=0x12345678;
pTest.ComData[1]=0x13579246;
pTest.ComData[2]=0x21436587;
pTest.ComData[3]=0x31427586;
```

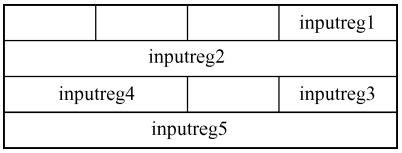


图 4-3　pTest 在内存中的摆放位置

通信协议中很少有这种规定的（空闲几个字节现象），除非通信双方故意约定，更多的是紧凑排列。通过将上面结构定义部分做一下改动，图 4-4 和图 4-5 是调整后的图。

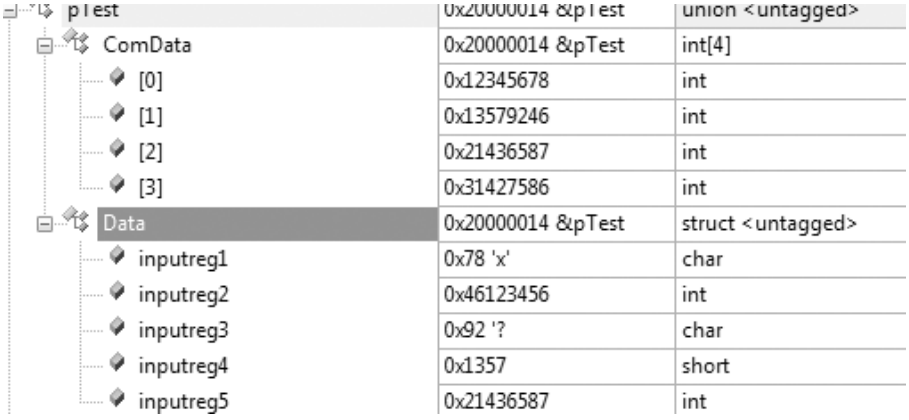


图 4-4　调整后的数据内存

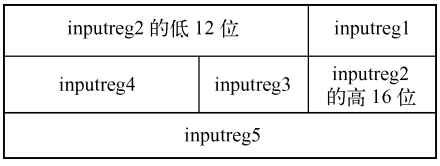


图 4-5　pTest 在内存中的新的摆放位置

```
#pragma push
#pragma pack(1)

typedef   union{
    int ComData[4]; //用于通信
  struct {
    char   inputreg1;
    int      inputreg2;
    char   inputreg3;
```

```
    short inputreg4;
  int   inputreg5;
}Data ;
} PData;
#pragma pop
```

#pragma pack(1)是设置结构体的边界对齐为 1 个字节，也就是所有数据在内存中是连续存储的，避免了 strcut 按照 ARM 的 32 位字节对齐，补齐 4 位字节的方式，如图 4-3 所示。

空间与效率的取舍，是根据应用场合确定的。当然也可通过调整变量定义位置调整。在图 4-3 中 inputreg3 与 inputreg4 之间空闲了一个字节，如果将 inputreg1、inputreg3、inputreg4 定义摆放在一起，也可节约一个字节空间。如何取舍取决于需求。

如果产品仅在一个 CPU 上运行，可随心所欲。如多 CPU 共存并需交互数据，需小心对待。如果两厂家产品采用通信方式配合，慎之再慎之，一定要沟通好数据格式，否则会有一种寸步难行的感觉，影响工期。

4.1.4 使用 volatile

在使用 C/C++语言开发嵌入式系统的时候，可能会遇到一些奇怪现象，例如：

- 打开编译器的编译优化选项，代码就不正常工作；
- 中断相关的程序结果时好时坏；
- 硬件驱动工作不稳定；
- 在轮询硬件时循环可能会卡住；
- 多任务系统中，单个任务工作正常，加入任何其他任务以后，系统就崩溃了。

这个时间需要将一些变量声明为 volatile。volatile 是一个变量声明限定词。它通知编译器，它所修饰的变量的值可能会在任何时刻被意外地更新，即便与该变量相关的上下文没有任何对其进行修改的语句。将变量声明为 volatile 会通知编译器该变量可以随时在执行之外进行修改，例如，由操作系统或硬件修改。因为 volatile 限定的变量值可以随时更改，所以只要在代码中引用变量，就一定会访问内存中变量的物理地址。这意味着编译器不能对变量执行优化，例如，将其高速缓存到本地寄存器中以避免内存访问。

MDK 的《编译器用户指南》推荐在以下情况时必须将变量声明为 volatile。

- 访问内存映射的外围设备；
- 多个线程之间共享全局变量；
- 在中断例程中访问全局变量；

4.1.5 RAM 中运行程序

有时，为了提高程序运行速度，会将某一段代码放进 RAM 中运行。MDK ARM 提供将一个编译单位放到 RAM 中运行的方法，做法如下。

（1）首先将计划在 RAM 运行的函数单独放到一个文件中。本例中新建的文件是 RamFunc.c，并在其中添加一个函数，函数名也为 RamFunc，可在 map 文件中观察该函数的区域。

（2）调整片内 RAM 分区，具体见图 4-6 的标注部分。

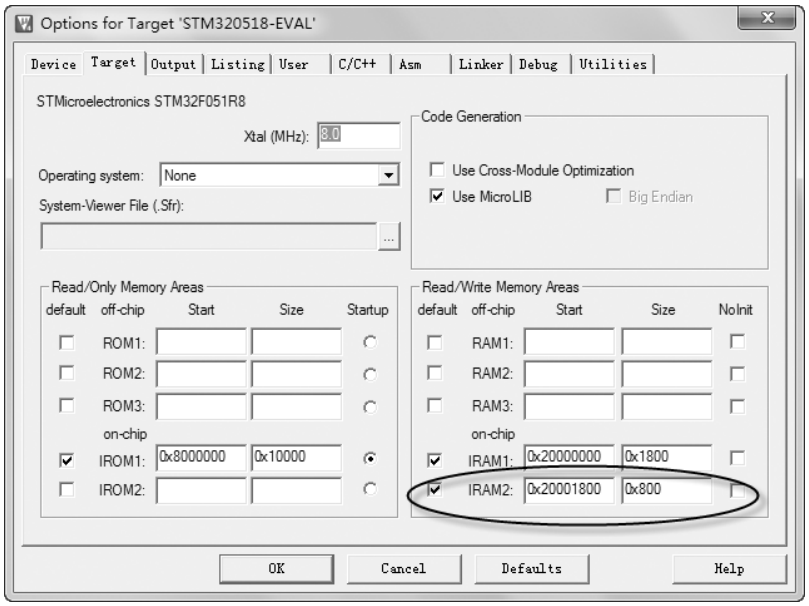


图 4-6　RAM 分区调整

（3）修改 RamFunc.c 的存储分配区域。选中 RamFunc.c 文件，单击右键，在弹出的菜单中选中“Options for File 'RamFunc.c'…”，如图 4-7 所示。在新出现的窗口中修改 Memory Assignment 中的 Code/Const 内容，如图 4-8 所示。

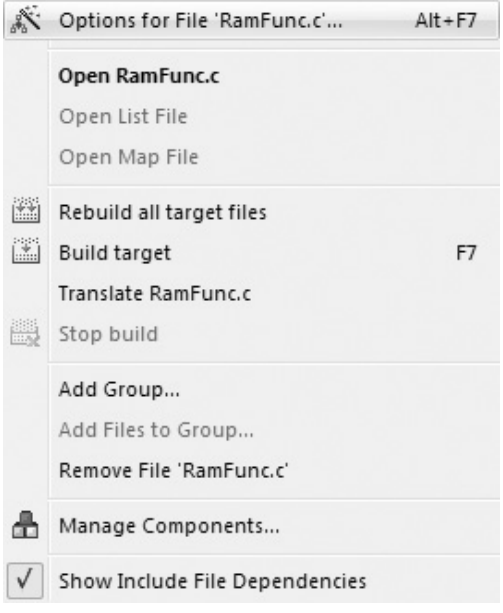


图 4-7　选择 RamFunc.c 文件修改

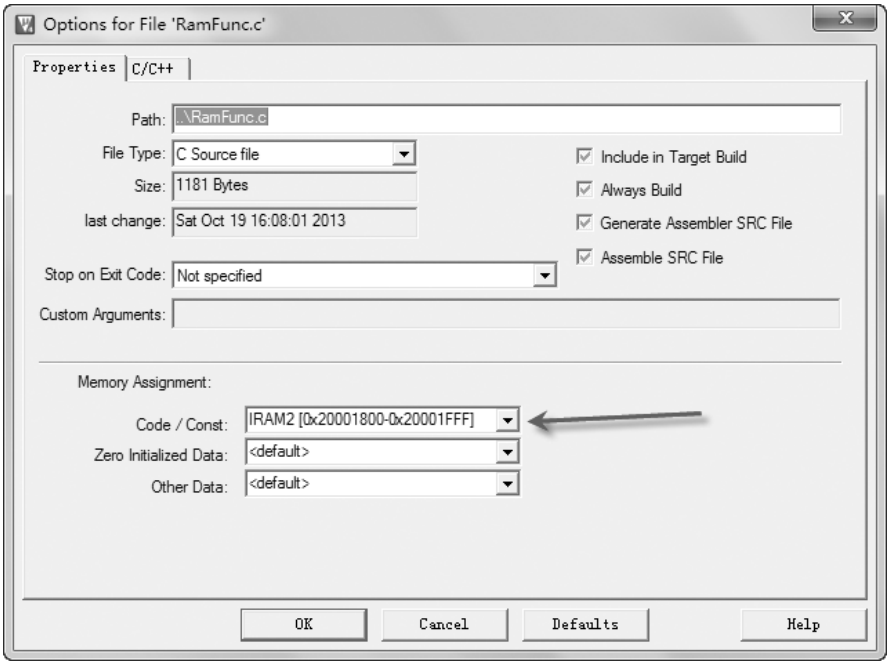


图 4-8　修改代码内存分配

编译链接工程后，通过检查 map 文件内容可以观察结果。其中有关包含 RamFunc 的 map 文件结果如下。

Symbol Name	Value	Ov Type	Size	Object(Section)
main	0x0800022b	Thumb Code	6	ramfunction.o(i.main)
Region$$Table$$Base	0x08000230	Number	0	anon$$obj.o(Region$$Table)
Region$$Table$$Limit	0x08000260	Number	0	anon$$obj.o(Region$$Table)
count	0x20000000	Data	4	ramfunction.o(.data)
__initial_sp	0x20000408	Data	0	startup_stm32f0xx.o(STACK)
RamFunc	0x20001801	Thumb Code	10	ramfunc.o(i.RamFunc)

从该文件中可以看到 RamFunc 在 0x20001801，是 Thumb 代码。

4.1.6　软件结构

嵌入式应用程序的结构主要有轮询方式、中断方式、轮询中断结合方式和实时操作系统方式。

轮询方式（也被称为超级循环模式）最简单，所有外设根据程序设计者思路不断查询状态，而后执行相应的任务。这种方式在实际产品中使用得比较少，多见于芯片厂家提供的例程，即通过相应外设寄存器状态，演示相应功能。例如，通过不断查询 ADC 的 EOC 状态位是否置位，从而判断 ADC 采样是否结束。该方式使得程序一直处于等待状态无法执行其他任务，CPU 利用率极为低下。在产品开发中几乎不采用该方式。

中断方式主要用于微处理器处于睡眠模式，通过外部资源或者片上外设产生的中断唤醒微处理器。该方式微处理器多数时间处于睡眠模式，即低功耗模式。在中断驱动应用程序

中，来自不同中断源的中断可设置不同优先级。高优先级中断可打断正运行的低优先级中断，从而降低高优先级中断源的延迟。该模式被大量用于一些具有定时唤醒功能的产品以及低功耗产品。中断方式的核心思想是以中断触发为核心进而处理逻辑。

超级循环与中断驱动组合模式，是指程序中存在大循环，也存在着中断程序。该种程序结构较为常见。中断程序用于响应时间要求高的任务。中断与 while(1)中的信息交互是通过全局变量方式进行的。

用于嵌入式微处理器的操作系统主要是嵌入操作系统（embedded OS）或者实时操作系统（Realtime OS）。二者最大的区别是对任务执行时间的确定性，不是体现在任务的执行速度上。第 16 章说明了实时操作系统如何处理任务以及中断。

4.2 CMSIS

CMSIS（Cortex Microcontroller Software Interface Standard）是 Cortex-M 系列处理器的硬件抽象层。CMSIS 紧密配合不同芯片厂商和软件厂商，并提供一个公共的接口供外设、操作系统、中间件使用。CMSIS 是为了能够融合多个中间件厂商的软件组件而设计的。CMSIS 提供了持续简单的软件接口，简化了软件重用性，降低了微处理器开发人员的学习曲线。

4.2.1 CMSIS 主要构成

图 4-9 是 CMSIS 的结构。

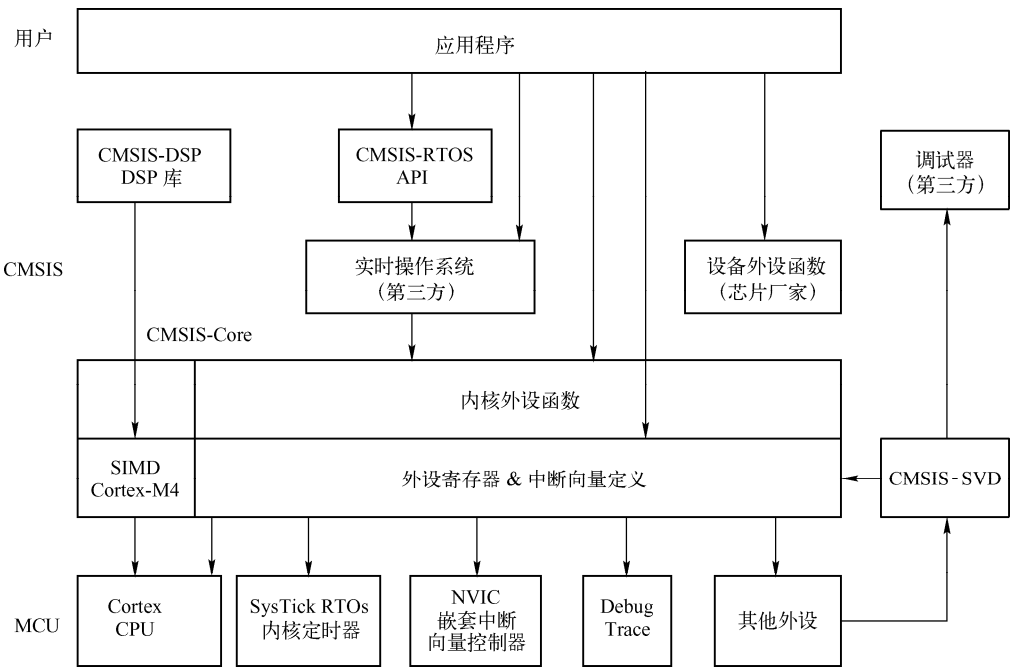


图 4-9　CMSIS 结构

❑ CMSIS-CORE：Cortex-M 处理器的内核和外设 API。提供了针对 Cortex-M0、Cortex-

M3、Cortex-M4、SC000 与 SC300 处理器的标准接口，也包含 Cortex-M4 的 SIMD 内置函数。

- CMSIS-DSP：DSP 库包含可 60 多个函数，可用于定点（小数的 q7, q15, q31）和单精度浮点数（32 位）。可用于 Cortex-M0，Cortex-M3 与 Cortex-M4 处理器。Cortex-M4 采用了 SIMD 指令集进行了优化。
- CMSIS-RTOS API：实时操作系统的公共 API。提供了标准化编程接口，可在不同 RTOS（实时操作系统）间移植，因此软件模板、中间件、库等其他组件也可在不同 RTOS 之间进行移植。

CMSIS-SVD：外设系统视图描述，以 XML 文件形式描述了外设。

CMSIS-CORE 内核实现了针对 Cortex-M 设备的基本运行时系统，允许用户使用处理器内核和设备外设。主要定义了如下内容。

- 硬件抽象层（HAL）：硬件抽象层提供了针对 SysTick、NVIC、系统控制块寄存器、MPU 寄存器、FPU 寄存器与内核存取函数的标准化定义。
- 系统异常名：系统异常的接口，无兼容性问题。
- 头文件的组织方法：使得学习新的 Cortex-M 更加容易，方便软件移植性，包含设备专用中断。
- 系统初始化的方法：被用于每一个 MCU 厂商，例如标准化的SystemInit()函数是必须的，该函数用于配置设备的时钟。
- 内在函数：标准 C 语言不包含的 CPU 指令函数。
- 定义系统时钟频率变量：简化配置 SysTick 定时器。

4.2.2 使用 CMSIS

为了使用 CMSIS-Core 需要在应用程序中添加下列文件。

- 启动文件 startup_<device>.s：复位异常向量。
- 系统配置文件 system_<device>.c 与 system_<device>.h：通用设备配置（例如时钟和总线设置）。
- 设备头文件 <device.h>：提供访问处理器内核和所有外设的方式。

注意：复位后执行启动文件 startup_<device>.s 并调用 SystemInit。在系统初始化后，由 C/C++运行时库控制，该库执行初始化并调用 main 函数。启动文件 startup_<device>.s 包含所有异常和中断向量，并实现了每个异常的默认函数，也包含栈和堆的配置。

系统配置文件 system_<device>.c 和 system_<device>.h 设置处理器时钟，变量 SystemCoreClock 代表 CPU 时钟速率。另外，该文件可能包含存储器总线设置和时钟的重配置。

设备头文件<device.h>是中心包含文件，提供了如下特征。

- 外设存取：所有外设的寄存器存取。也可能包含设备专用外设的函数。
- 中断和异常（NVIC）：嵌套中断向量控制器的标准符号和函数。
- CPU 指令的封装函数：用于访问专用指令。
- SysTick 定时器（SYSTICK）：用于配置和启动周期定时器中断。

CMSIS-CORE（见图 4-10）是设备专用的。启动文件 startup_<device>.s 是编译器专用的。不同的编译厂商工具链可能提供各种支持设备的目录。

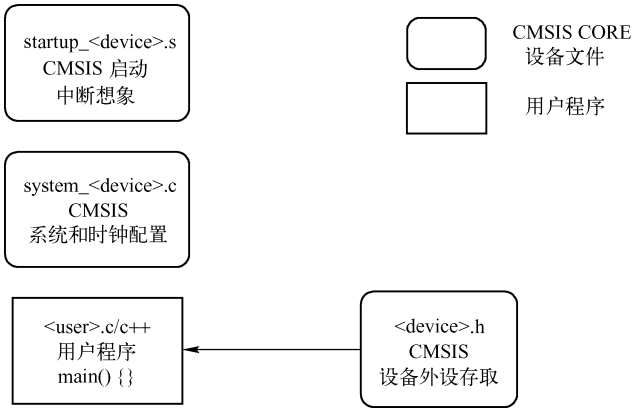


图 4-10　CMSIS-CORE 用户文件结构

4.3　STM32F0xx 标准外设库

标准外设库支持 STM32F0xx 系列产品。目前标准外设库版本是 1.3（在 ST 公司网站的 Part Number 是 **STSW-STM32048**），下载地址是 http://www.st.com/web/en/catalog/tools/PF257884或者http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/PF257884

注意：由于 ST 公司的外设库会不断升级。在本书开始写作过程中，最开始见到的是 1.2 版本，很快就发展到了 1.40 版本。本书是根据 V1.31 版本编写的。具体后续版本的差异请对照外设库的说明。可使用 WinMerge 软件通过目录比对方式做比较。目前 STM32F0xx 的几个 discovery 版本是均是基于外设库 1.2 版本。

4.3.1　标准外设库概述

STM32F0xx 标准外设库有如下 3 个抽象层。

- ❑ 完整的寄存器地址映射：采用 C 语言以位域方式定义了所有寄存器的位，避免寄存器需直接以地址方式操作。
- ❑ 覆盖了所有外设功能的程序和数据结构集合：可直接用作参考框架，也包含用于支持内核相关的固有特征、公共变量、数据类型的宏。
- ❑ 覆盖所有外设的例程集合：支持多数开发工具的模板工程，从而可以在极短时间进行新的芯片开发。

每一驱动程序由涵盖所有外设功能的函数集合构成。每个驱动程序开发使用公共 API 驱动，该 API 具有标准化的驱动结构、函数和参数名。

驱动程序源代码以 Strict ANSI-C 开发，符合 MISRA-C 2004 标准。因采用 Strict ANSI-C 书写整个库函数，所以整个库独立于软件工具链，只有启动文件依赖工具链。

标准外设库实现了通过检查所有库函数的输入参数进行运行时故障检测。这种动态监测

增强了软件的健壮性。运行时检测适合用于用户应用程序开发和调试，程序开发完毕后可通过宏定义将检测代码屏蔽，从而减少代码量并提高执行速度。

标准外设通用型决定了代码量和执行速度不是最优的。对代码尺寸和/或代码执行速度有严格要求的应用程序，库驱动程序应该被作为一个如何配置外设的参考，需要根据应用程序要求进行裁剪。

注意：应用程序的代码性能与代码量和或速度有关，同时也与 C 语言编译器的优化设置有关。为了使得应用程序更好、更快，或者代码量与速度之间的折中选择，需要根据应用程序需求进行微调优化。

外设库采用模块化编程方式实现的，确保在不同组件间构建应用程序的独立性，保证了移植性。图 4-11 提供了 STM32F0xx 标准外设库的用法以及与其他组件之间的关系。

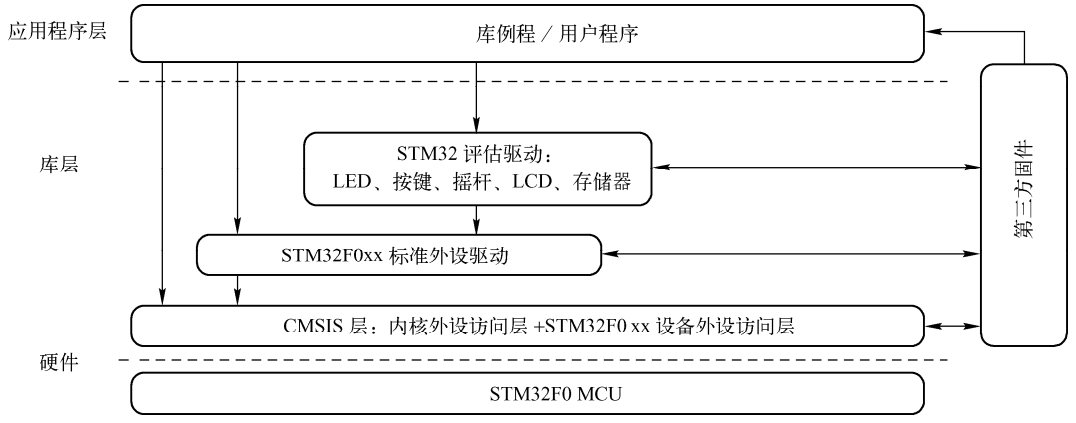


图 4-11　STM32F0 固件库框架

4.3.2　STM32F0xx 外设驱动文件说明

标准外设库文件的关系如图 4-12 所示。

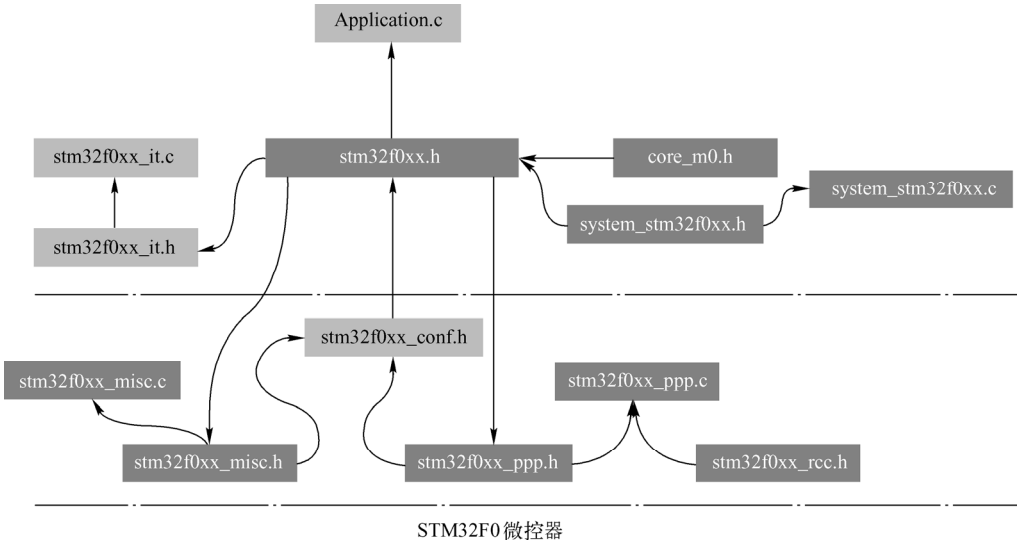


图 4-12　固件库的文件关系

每个外设会对应文件源代码文件 stm32f0xx_ppp.c 和一个头文件 stm32f0xx_ppp.h。其中，stm32F0xx_ppp.c 包含了 PPP 外设所包含的固件函数。

所有外设都需包含内存映射文件，即 stm320f0xx.h。用户程序只需包含该文件，在第 2 章的 helloworld 工程中已经见到此用法。

stm32f0xx.h 文件包含了 STM32F0xx 的所有外设寄存器定义、位定义和内存映射，包含以下内容。

- IRQ 通道定义。
- 外设内存映射和物理寄存器地址定义。
- 外设指针申明和驱动程序头文件包含。
- 芯片的各种配置，例如外设晶振（HSE）值。

表 4-1 列出了外设库的不同文件以及功能说明。

表 4-1　固件库文件的描述

文　件　名	描　　述
stm32f0xx_conf.h	外设的驱动程序配置文件。 通过修改文件中所包含的外设头文件，用户可启动或者禁用外设驱动。 编译固件库的驱动程序之前，定义USE_FAULT_ASSERT，通过预处理，可用于启用或禁用库的运行时故障检测
stm32f0xx_ppp.h	ppp的外设头文件，这里的ppp只是一个代码，在实际中是具体的外设名字，例如ADC、DMA等。 该文件包含ppp外设功能和使用这些功能的变量定义
stm32f0xx_ppp.c	用C语言编写的ppp外设驱动程序的源代码实现文件
stm32f0xx_it.h	头文件，包含所有的外设中断处理程序原型
stm32f0xx_it.c	中断源文件模板，Cortex-M0异常中断服务程序例程（ISR），添加相应外设的ISR程序（可用外设的中断处理程序的名称，具体代码请参阅启动文件startup_stm32f0xx.s）

4.3.3　STM32F0xx 的 CMSIS 文件说明

表 4-2 是 STM32F0xx CMSIS 文件的说明。

表 4-2　STM32F0xx CMSIS 文件的说明

文　件　名	描　　述
stm32f0xx.h	CMSIS Cortex-M0 STM32F0xx 的设备外设访问层的头文件。该文件是唯一的包含文件，程序员使用 C 源代码，通常在 main.c 应用。该文件包含以下内容。 ❑ 允许选择配置。 ➢ 在目标应用中使用的设备。 ➢ 使能或失能在应用程序代码中的外设设备的驱动程序（即代码将基于访问外设的寄存器，而不是驱动 API），是通过控制#define USE_STDPERIPH_DRIVER 实现的。 ➢ 改变应用程序的几个特定参数，如 HSE 晶体频率。 ❑ 数据结构和所有外围设备的地址映射。 ❑ 外设的寄存器申明和位定义。 ❑ 通过宏访问外设的寄存器的硬件。 ❑ IRQ 通道定义
system.stm32f0xx.h	CMSIS Cortex-M0 STM32F0xx 设备外设访问层系统头文件
system.stm32f0xx.c	CMSIS Cortex-M0 STM32F0xx 设备外设访问层系统源文件
startup_stm32f0xx.s	STM32F0xx 设备启动文件，每一种编译器对应一个文件

由于 STM32F0 芯片之间的差异，存在多个 startup_stm32F0xx.s 文件，如 startup_stm32f030.s 与 startup_stm32f051.s。启动文件间的主要差异是中断向量定义。图 4-13 是通过 WinMerge 软件将 startup_stm32f030.s 与 startup_stm32f051.s 两个文件进行比对的结果。

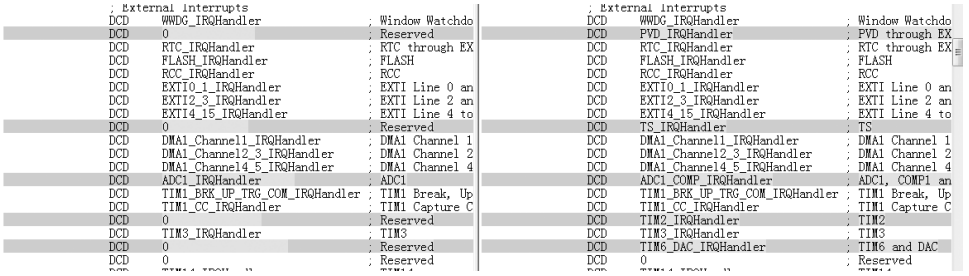


图 4-13　startup_stm32f030.s 与 startup_stm32f051.s 对比

4.3.4　库文件夹说明

STM32F0xx 标准外设库是一个 zip 文件，解压该文件会生成文件夹 STM32F0xx_StdPeriph_Lib_VX.Y.Z，该文件夹包含了标准外设库的所有文件。图 4-14 是库文件的组织结

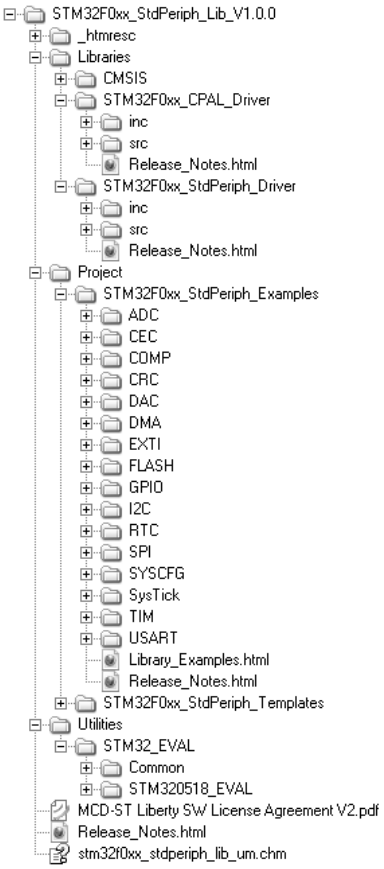


图 4-14　文件组织结构

构，该文件夹包含了所有 CMSIS 文件和标准外设的驱动程序。

（1）_htmresc 目录包含了所有包的 html 资源。

（2）Libraries 目录包含了所有的 CMSIS 文件和 STM32F0xx 标准外设的驱动程序。

- CMSIS 子文件夹包含了 STM32F0xx 的 CMSIS 文件，用于设备外设访问层和内核外设访问层。
  - STM32F0xx_CPAL_Drive 子文件夹包含了 STM32F0xx 的 CPAL 文件，用于 I^2C 外设的 API。
  - STM32F0xx_StdPeriph_Driver 子文件夹包含了构成库的核心的所有子目录和文件。
  - inc 子目录包含了外设驱动程序头文件，不需用户进行修改。
  - stm32f0_misc.h：各种固件库函数的函数原型。
  - stm32f0xx_ppp.h（一个外设对应一个头文件）是函数原型、数据结构、枚举数据。
  - src 子文件夹包含了所有外设的驱动程序源代码，不需用户进行修改。
  - stm32f0_misc.c：各种固件库函数（附件在 CMSIS 函数上的）。
  - stm32f0xx_ppp.c（一个外设对应一个源代码文件）：每个外设的函数主体。

（3）Project 目录包含了模板工程和 STM32F0xx 标准外设例程。

- STM32F0xx_StdPeriph_Examples 子目录是所有例子的完整目录。该目录中的子文件夹是运行每个外设例程所必需的文件。每个子目录主要有以下内容。
  - readme.txt：描述例程以及如何使用。
  - stm32f0xx_conf.h：使能/停用外设驱动程序的头文件。
  - stm32f0xx_it.c：中断句柄的源文件（函数主体可能是空的）。
  - stm32f0xx_it.h：所有中断句柄原型的头文件。
  - main.c：例程代码。
  - system_stm32f0xx.c：该文件提供了设置 STM32F0 系统的函数，主要有时钟配置、AHB/APBx 预分频和 Flash 设置。
- STM32F0xx_StdPeriph_Templates 子目录包含了标准模板工程，可被 EWARM、MDK-ARM、TASKING 和 TrueSTUDIO 工具链编译。
  - stm32f0xx_conf.h：配置头文件。
  - stm32f0xx_it.c：包含中断服务程序的源文件（在该模板中函数体是空的）。
  - stm32f0xx_it.h：各种中断服务程序的函数原型的头文件。
  - main.h：main.c 的头文件。
  - main.c：主程序。
  - system_stm32f0xx.c：用于设置 STM32F0 系统的函数，配置系统时钟、AHB/APBx 的预分频和 Flash 设置。

4.3.5　固件库文件

位于 STM32F0xx_StdPeriph_Driver 文件下面的 inc 和 src 两个文件夹是关于 STM32F0xx 外设的各种驱动程序的文件夹。src 是各个设备驱动成的源文件。inc 为对应 src 文件中的头

文件。主要有以下内容。

（1）stm32f0xx_adc.c 主要提供了 ADC 的管理功能

- 初始化和配置
- 省电模式
- 模拟看门狗配置
- 温度传感器、Vrefint（内部参考电压）与 Vbat（电池电压）管理
- ADC 通道配置
- 规则通道 DMA 配置
- 注入通道的配置
- 中断和标志管理

（2）stm32f0xx_cec.c 文件提供了消费电子控制接口的管理函数

- 初始化和配置
- 数据传输功能
- 中断和标志管理

（3）stm32f0xx_comp.c 文件提供了比较器（COMP1 和 COMP2）外设的管理功能

- 比较器配置
- 窗模式控制

（4）stm32f0xx_crc.c 文件提供了所有的 CRC 固件功能

- CRC 的配置
- 一个/多个 32 位数据的 CRC 计算
- CRC 的独立寄存器（IDR）访问

（5）stm32f0xx_dac.c 文件提供了如下 DAC 外设管理功能

- DAC 通道配置包括触发方式、输出缓冲区、数据格式
- DMA 管理
- 中断和标志管理

（6）stm32f0xx_dbgmcu.c 提供了所有 DBGMCU 固件功能

- 设备和版本管理
- 外设配置

（7）stm32f0xx_dma.c 文件提供了如下直接存储器存取控制器（DMA）功能

- 初始化和配置
- 数据计数器
- 双缓存模式配置和命令
- 中断和标志管理

（8）stm32f0xx_exti.c 文件提供了如下 EXTI 外设管理功能

- 初始化和配置
- 中断和标志管理

（9）stm32f0xx_flash.c 文件提供了如下 Flash 外设管理功能

- Flash 接口配置

- Flash 存取器编程
- 字节选项编程
- 中断和标志管理

（10）stm32f0xx_gpio.c

- 初始化和配置
- GPIO 的读/写
- GPIO 配置备用功能

（11）stm32f0xx_i2c.c 提供了 I^2C 管理功能

- 初始化配置
- 数据传输
- SMBUS 管理
- DMA 传输管理
- 中断事件和标志管理

（12）stm32f0xx_iwdg.c

- 预分频器和计数器配置
- IWDG 激活
- 标志管理

（13）stm32f0xx_rcc.c 提供了复位和时钟控制（RCC）的管理功能

- 内部/外部时钟、PLL、CSS 和 MCO 配置
- 系统 AHB 和 APB 总线时钟配置
- 外设时钟配置
- 中断和标志管理

（14）stm32f0xx_rtc.c [code]提供了实时时钟的管理功能

- 初始化
- 日历（时间和日期）配置
- 报警（报警 A 和报警 B）配置
- 唤醒定时器配置
- 夏令时配置
- 输出引脚配置
- 时间戳配置
- 备份数据寄存器配置
- RTC 的篡改、TimeStamp 选型和输出类型配置
- 中断和标志管理

（15）stm32f0xx_spi.c 提供了串行通信接口（SPI）的管理功能

- 初始化和配置
- 数据传输功能
- 硬件 CRC 计算

- ❑ DMA 传输管理
- ❑ 中断和标志管理

（16）stm32f0xx_tim.c 文件提供了 TIM 的管理功能

- ❑ 时基管理
- ❑ 输出比较管理
- ❑ 输入捕捉管理
- ❑ 高级控制定时器（TIM1 和 TIM2）的特定功能
- ❑ 中断、DMA 和标志管理
- ❑ 时钟管理
- ❑ 特定的接口管理
- ❑ 具体的映射管理

（17）stm32f0xx_usart.c 文件描述了通用同步/ 异步收发器（USART）的管理功能

- ❑ 初始化和配置
- ❑ 数据传输
- ❑ 多处理器通信
- ❑ LIN 模式
- ❑ 半双工模式
- ❑ 智能卡模式
- ❑ IrDA 模式
- ❑ DMA 传输管理
- ❑ 中断和标志管理

（18）stm32f0xx_wwdg.c 文件提供了系统看门狗的管理功能

- ❑ 预分频器、刷新窗口和计数器配置
- ❑ WWDG 激活
- ❑ 中断和标志管理中断向量表的定义

4.3.6 MDK ARM 中使用固件库实例

STM32F0xx_StdPeriph_Templates 目录提供了模板，由于每个例程在不同目录下面，需要复制 stm32f0xx_conf.h 、 stm32f0xx_it.c: 、 stm32f0xx_it.h 、 main.h: main.c 、 system_stm32f0xx.c 到 STM32F0xx_StdPeriph_Templates 目录才可执行。

以固件库中 Projects\STM32F0xx_StdPeriph_Examples\SysTick 例子说明该过程。

（1）进入 STM32F0xx_StdPeriph_Examples\SysTick\SysTick_Example 目录。

（2）复制所有文件到 STM32F0xx_StdPeriph_Templates 目录下。

（3）打开该目录下的 MDK-ARM 中的 Project.uvproj。

在打开的 MDK 工程中，单击 project→manage→Components,Environment,Books…，出现如图 4-15 所示对话框。将相应的 Project Targes 设置为当前 Target。本书中主要是设置成 STM32F030。

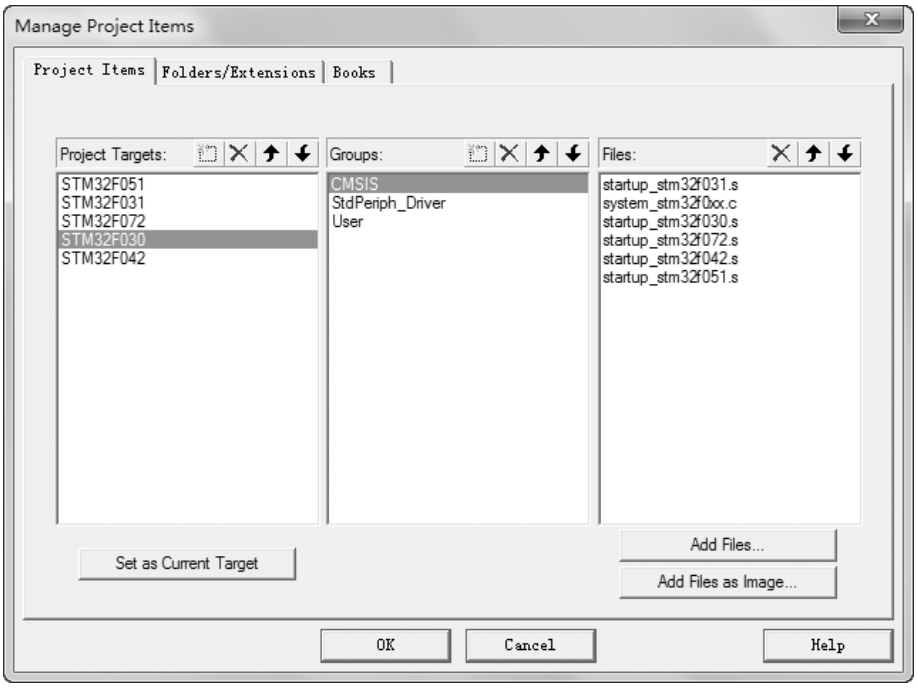


图 4-15　固件库在 MDK 中使用

4.4　小　　结

本章首先介绍了嵌入式软件开发时 C 语言的特殊之处，以及如何在 ARM 中操作内部寄存器等内容，然后介绍了由 ARM 公司提出的 CMSIS 标准，最后介绍了 ST 公司针对 STM32F0x 系列芯片的固件库结构。

第5章

通用 I/O（GPIO）

STM32F0 的 I/O 引脚通过一个多路复用器连接到片上外设，同一时间一个引脚只允许一种复用功能。每个引脚对应八种输入复用（AF0～AF7）的多路复用器，使得每个引脚可能对应多种外设。故可根据产品设计要求最优地配置引脚，方便电路板设计，并且 STM32F0 的 GPIO 挂靠在 AHB 总线上具有更高速度。

5.1 GPIO 引脚与功能

5.1.1 引脚描述

STM32F030 的封装形式主要有 TSSOP20、LQFP32、LQFP48、LQFP64，STM32F050 的封装形式主要有 LQFP48、UFQFPN32、TSSOP20、UFQFPN28，STM32F051 的封装形式主要有 LQFP64、UFQFPN32、LQFP48、LQFP32，STM32F072xx 的封装形式主要有 LQFP100、LQFP64、LQFP48、UFQFPN48、WLCSP49、UFBGA100。

STM32F0x 的一些 I/O 引脚可能是输入标准 3.3V 的引脚（标注为 TC），也可能是耐受 5V 输入（标注为 FT）引脚以及 3.3V 直接连到 ADC 的引脚（标注 TTa）。有关引脚耐压情况可查阅各芯片的数据表中芯片定义表格中的 I/O structure 一栏。

GPIO 是通用型输入/输出（General Purpose I/O）的简称，功能类似 8051 的 P0～P3，其接脚可以供使用者由软件控制使用，引脚可作为通用输入（GPI）、通用输出（GPO）或通用输入与输出（GPIO）等。对于输入，可以通过读取某个寄存器来确定引脚电平的高低；对于输出，可通过写入某个寄存器来让这个引脚输出高电平或者低电平；对于其他特殊功能，则有另外的寄存器来控制它们。

STM32F0 系列每个通用 I/O 口都有 4 个 32 位配置寄存器（GPIOx_MODER、GPIOx_OTYPER、GPIOx_OSPEEDR 和 GPIOx_PUPDR）、两个 32 位数据寄存器（GPIOx_IDR 与 GPIOx_ODR）和 1 个 32 位置位/复位寄存器（GPIOx_BSRR）。端口 A 和端口 B 还含有 1 个 32 位锁定寄存器（GPIOx_LCKR）和两个 32 位复用功能寄存器（GPIOx_AFRH 与 GPIOx_AFRL）。STM32F07X 的端口 C、D 和 E 有两个 32 位复用功能寄存器（GPIOx_AFRH 余 GPIOx_AFRL）。端口 E 仅仅在 STM32F07X 系列芯片上可用。

注意：STM8 的每个端口都分配有一个输出数据寄存器、一个输入引脚寄存器、一个数据方向寄存器、一个选择寄存器和一个配置寄存器。STM8 的寄存器是 8 位的，STM32F0xx 的寄存器是 32 位的。

表 5-1 是 ST32F0 各类型的 GPIO 数量，在设计电子琴、PLC 和开关量需要比较多 I/O 设备时，可根据表 5-1 选择相应型号的芯片。

表 5-1　STM32F0x 的 GPIO 数量

芯片型号	STM32F051Kx	STM32F051Cx	STM32F051Rx	STM32F050Fx	STM32F050Gx	STM32F050Kx	STM32F050Cx
GPIO 数量	25 (LQFP32) 27 (UFQFPN32)	39	55	15	23	27	39
芯片型号	STM32F030F4	STM32F030K6	STM32F030C6/C8	STM32F030R8	STM32F072Cx	STM32F072Rx	STM32F072Cx
GPIO 数量	15	26	39	55	37	51	87
芯片型号	STM32F042Fx	STM32F042G	STM32F042K	STM32F042T	STM32F042C		
GPIO 数量	16	24	26/28	30	38		

GPIO 具有如下特点。

- ❑ 输出状态：带有上拉或下拉的推挽输出或开漏输出。
- ❑ 从数据寄存器（GPIOx_ODR）或外设（复用功能输出）输出数据。
- ❑ 可选的每个 I/O 口的速度。
- ❑ 输入状态：浮空、上拉/下拉、模拟输入。
- ❑ 从数据寄存器（GPIOIDR）或外设输入数据（复用功能输出）。
- ❑ 位置位/复位寄存器（GPIOx_RR）为对 GPIOx_ODR 寄存器提供位访问能力。
- ❑ 端口 A 或端口 B 的锁定机制（GPIOx_LCKR）配置。
- ❑ 模拟功能。
- ❑ 可选的端口 A 和端口 B 复用功能。
- ❑ 每两个时钟周期快速切换口线值能力。
- ❑ 允许 GPIO 口和外设引脚的高灵活性复用。

5.1.2　GPIO 功能描述

GPIO 端口的每个位可以由软件配置成浮空输入、上拉输入、下拉输入、模拟输入、具有上拉或下拉能力的开漏输出、具有上拉或下拉能力的推挽输出、复用功能且具有上拉或下拉能力的推挽输出、复用功能且具有上拉或下拉能力的开漏输出。

小知识：推挽输出（英语：Push–pull output）是指两个参数相同的功率三极管或MOSFET管，以推挽方式存在于电路中。因为元件受到两个互补信号的制约，总会保持一个导通，一个截止的状态。推拉式输出级既提高电路的负载能力，又提高开关速度。

开漏输出：输出端相当于三极管的集电极，要得到高电平状态需要上拉电阻才行，适合于做电流型的驱动，其吸收电流的能力相对强（一般 20mA 以内）。开漏输出具有利用外部电路的驱动能力，减少 IC 内部的驱动。一般来说，开漏是用来连接不同电平的器件，匹配电平用的，因为开漏引脚不连接外部的上拉电阻

时，只能输出低电平，如果需要同时具备输出高电平的功能，则需要接上拉电阻，很好的一个优点是通过改变上拉电源的电压，便可以改变传输电平。比如，加上上拉电阻就可以提供 TTL/CMOS 电平输出等。（上拉电阻的阻值决定了逻辑电平转换的沿的速度。阻值越大，速度越低功耗越小，所以负载电阻的选择要兼顾功耗和速度。）

每个 I/O 端口位可以自由编程，但 I/O 端口寄存器可按 32 位字、半字或字节访问。GPIOx_BSRR 寄存器允许对任何 GPIO 寄存器进行位读/改写访问。在这种情况下，在读和更改访问之间产生 IRQ 时也不会发生危险。

图 5-1 给出了一个标准 I/O 端口位。5V 耐压 I/O 端口与图 5-1 的区别在保护二极管处的电压。表 5-2 给出了端口位的可能配置。

表 5-2　端口位配置表

<table>
<tr><th>MODER [1:0]</th><th>OTYPER</th><th colspan="2">OSPEEDR [B:A]</th><th colspan="2">PUPDR [1:0]</th><th colspan="2">I/O配置</th></tr>
<tr><td rowspan="8">01</td><td>0</td><td colspan="2" rowspan="8">SPEED [B:A]</td><td>0</td><td>0</td><td>GP输出</td><td>PP</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>GP输出</td><td>PP + PU</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>GP输出</td><td>PP + PD</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td colspan="2">保留</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>GP输出</td><td>OD</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>GP输出</td><td>OD + PU</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>GP输出</td><td>OD + PD</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td colspan="2">保留（GP输出OD）</td></tr>
<tr><td rowspan="8">10</td><td>0</td><td colspan="2" rowspan="8">SPEED [B:A]</td><td>0</td><td>0</td><td>AF</td><td>PP</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>AF</td><td>PP + PU</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>AF</td><td>PP + PD</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td colspan="2">保留</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>AF</td><td>OD</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>AF</td><td>OD + PU</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>AF</td><td>OD + PD</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td colspan="2">保留</td></tr>
<tr><td rowspan="4">00</td><td>x</td><td>x</td><td>x</td><td>0</td><td>0</td><td>输入</td><td>浮空</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>0</td><td>1</td><td>输入</td><td>PU</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>1</td><td>0</td><td>输入</td><td>PD</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>1</td><td>1</td><td colspan="2">保留（浮空输入）</td></tr>
<tr><td rowspan="4">11</td><td>x</td><td>x</td><td>x</td><td>0</td><td>0</td><td>输入/输出</td><td>模拟</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>0</td><td>1</td><td colspan="2" rowspan="3">保留</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>1</td><td>0</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>1</td><td>1</td></tr>
</table>

注：GP =通用，PP =推挽输出，PU =上拉，PD =下拉，OD =开漏，AF =复用功能。

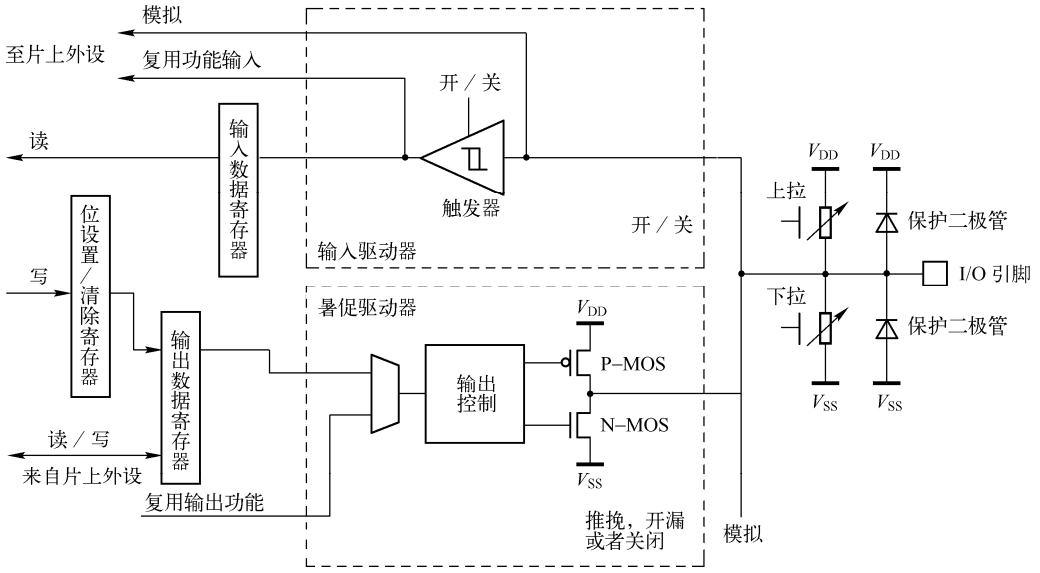


图 5-1　标准 I/O 端口位的基本结构

5.1.3　通用 I/O（GPIO）

复位期间和刚复位后，复用功能未开启且所有的 I/O 端口处于浮空输入模式。复位后，调试引脚被置为复用功能的上拉/下拉模式。

- PA14：SWCLK 置于下拉模式。
- PA13：SWDAT 置于上拉模式。

当作为输出配置时，写到输出数据寄存器（GPIOx_ODR）的值输出到相应的引脚上。可以以推挽模式或开漏模式（仅低电平被驱动，高电平表现为高阻）使用输出驱动器。

输入数据寄存器（GPIOx_IDR）在每个 AHB 时钟周期捕捉 I/O 引脚上的数据。所有 GPIO 引脚都有一个内部弱上拉和弱下拉电阻，它们被激活或断开取决于 GPIOx_PUPDR 寄存器。

5.1.4　I/O 引脚的复用功能和重映射

器件 I/O 口线通过多路复用器连接到内嵌的外设/模块。同一口线上不能有冲突的外设引脚分配。每个 I/O 引脚有一个多达 16 个复用功能输入（AF0～AF15）的多路复用器，其可通过配置 GPIOx_AFRL 寄存器（引脚 0～引脚 7）和 GPIOx_AFRH 寄存器（引脚 8 ～引脚 15）来实现（见 5.1.12）。

- 复位后，所有的 I/O 口都连接到复用功能 0（AF0）。
- 有关每个引脚的具体复用功能在器件数据手册有详尽描述。

除了这种灵活的 I/O 复用结构，每个外设还有复用功能映射到不同的 I/O 引脚上，这种方法用于在小封装器件上优化，获得更多的可用外设。

为使用一个给定的 I/O 口配置，需遵守如下事项。

- 调试功能：每个器件复位后，这些引脚立即配置为复用功能用来支持调用。

- GPIO：在 GPIOx_MODER 寄存器中配置所需的 I/O 口为输出、输入或模拟输入。
- 外设的复用功能：
  - 连接 I/O 到所需的 AFx，AFx 定义在 GPIOx_AFRL 或 GPIOx_AFRH 寄存器中。
  - 通过对 GPIOx_OTYPER、GPIOx_PUPDR 和 GPIOx_OSPEEDER 寄存器来配置相应引脚的上拉/下拉和输出速度。
  - 在 GPIOx_MODER 寄存器中配置所需的 I/O 口线为复用功能。
- 附加功能：
  - 对于 ADC 和 DAC，在 GPIOx_MODER 寄存器中配置所需的 I/O 口线为模拟方式并在 ADC 或 DAC 寄存器中配置所需的功能。
  - 对于附加功能如 RTC、WKUPx 和振荡器，在关联的 RTC、PWR 和 RCC 寄存器配置相应所需的功能。

5.1.5 外部中断/唤醒线

所有端口都有外部中断能力。为了用作外部中断口线，端口线必须配置为输入模式（见第 6 章）。

5.1.6 输入配置

当 I/O 口配置为输入时：

- 该输出缓冲区禁用。
- 施密特触发器输入激活。
- 由 GPIOx_PUPDR 寄存器的值来激活上拉和下拉电阻。
- 在每个 AHB 时钟周期，I/O 引脚上的数据被采样进入输入数据寄存器。
- 用对输入数据寄存器的读访问来获取 I/O 口状态。

5.1.7 输出配置

当 I/O 口配置为输出时：

- 输出缓冲开启。
  - 开漏模式：输出寄存器上的“0”激活 N-MOS，而输出寄存器上的“1”将端口置于高阻状态（P-MOS 从不被激活）。
  - 推挽模式：输出寄存器上的“0”激活 N-MOS，而输出寄存器上的“1”将激活 P-MOS。
- 施密特触发输入被激活。
- 弱上拉和弱下拉电阻是否激活取决于 GPIOx_PUPDR 寄存器的值。
- 在每个 AHB 时钟周期，I/O 引脚上的数据被采样进入输入数据寄存器。
- 用对输入数据寄存器的读访问来获取 I/O 口状态。
- 用对输出寄存器的读访问来获取最后写进该寄存器的值。

5.1.8 复用功能配置

当 I/O 端口被配置为复用功能时：

- 在开漏或推挽模式下输出缓冲器可被配置。
- 外设的信号驱动输出缓冲器。
- 施密特触发输入被激活。
- 弱上拉和弱下拉电阻是否激活取决于 GPIOx_PUPDR 寄存器的值。
- 在每个 AHB 时钟周期，I/O 引脚上的数据被采样进入输入数据寄存器。
- 用对输入数据寄存器的读访问来获取 I/O 口状态。

5.1.9　模拟配置

当 I/O 端口为模拟配置时：

- 输出缓冲器关闭。
- 禁止施密特触发输入，实现了每个模拟 I/O 引脚上的零消耗。施密特触发输出值被强置为 0。
- 弱上拉和下拉电阻被禁止。
- 读取输入数据寄存器时数值为 0。

5.1.10　HSE 或 LSE 引脚用作 GPIO

当 HSE 或 LSE 振荡器关断时（复位后的默认状态），相关振荡器引脚可以用作普通的 GPIO 口。当 HSE 或 LSE 振荡器开启（在 RCC_CSR 寄存器设置 HSEON 或 LSEON 位来开启），振荡器控制其相关引脚且相关引脚的 GPIO 配置无效。当振荡器配置为用户外部时钟方式，仅使用 OSC_IN 或 OSC32_IN 引脚作为时钟输入，OSC_OUT 或 OSC32_OUT 引脚可仍然配置为正常的 GPIO 引脚。

5.1.11　备份域供电下 GPIO 引脚的使用

当 VCORE 域断电（当器件进入待机模式）时，PC13/PC14/PC15 GPIO 功能失去。在该情况下，若这些 GPIO 配置为不被 RTC 配置旁路，这些引脚被设为模拟输入模式。

5.1.12　GPIO 复用功能寄存器

GPIO 复用寄存器分为 GPIOx_AFRL 低寄存器与 GPIOx_AFRH 高寄存器，其中 x = A, B。图 5-2 是 GPIOx_AFRL 低寄存器的位图，4 位为一组，代表 AF0～AF7。GPIOx_AFRL 低寄存器分别对应 GPIOx（x=0～7）引脚，GPIOx_AFRL 对应 GPIOy（y = 8～15）。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-2　GPIO 复用功能低位寄存器（GPIOx_AFRL）

STM32F030x4、STM32F030x6、STM32F030x8 通过 GPIOA_AFR 可选择 PA 系列引脚的复用功能，见表 5-3。通过 GPIOB_AFR 可选择 PB 系列引脚的复用功能，见表 5-4。

表 5-3　STM32F030x4、STM32F030x6、STM32F030x8 的 PA 引脚复用

引　脚	AF0	AF1	AF2	AF3	AF4	AF5	AF6
PA0	—	USART1_CTS USART2_CTS	—	—	—	—	—
PA1	EVENTOUT	USART1_RTS USART2_RTS	—	—	—	—	—
PA2	TIM15_CH1	USART1_TX USART2_TX	—	—	—	—	—
PA3	TIM15_CH2	USART1_RX USART2_RX	—	—	—	—	—
PA4	SPI1_NSS	USART1_CK USART2_CK	—	—	TIM14_CH1	—	—
PA5	SPI1_SCK	—	—	—	—	—	—
PA6	SPI1_MISO	TIM3_CH1	TIM1_BKIN	—	—	TIM16_CH1	EVENTOUT
PA7	SPI1_MOSI	TIM3_CH2	TIM1_CH1N	—	TIM14_CH1	TIM17_CH1	EVENTOUT
PA8	MCO	USART1_CK	TIM1_CH1	EVENTOUT	—	—	—
PA9	TIM15_BKIN	USART1_TX	TIM1_CH2	—	I2C1_SCL	—	—
PA10	TIM17_BKIN	USART1_RX	TIM1_CH3	—	I2C1_SDA	—	—
PA11	EVENTOUT	USART1_CTS	TIM1_CH4	—	—	—	—
PA12	EVENTOUT	USART1_RTS	TIM1_ETR	—	—	—	—
PA13	SWDIO	IR_OUT	—	—	—	—	—
PA14	SWCLK	USART1_TX USART2_TX	—	—	—	—	—
PA15	SPI1_NSS	USART1_RX USART2_RX	—	EVENTOUT	—	—	—

表 5-4　STM32F030x4、STM32F030x6、STM32F030x8 的 PB 引脚复用

引　脚	AF0	AF1	AF2	AF3
PB0	EVENTOUT	TIM3_CH3	TIM1_CH2N	—
PB1	TIM14_CH1	TIM3_CH4	TIM1_CH3N	—
PB2	—	—	—	—
PB3	SPI1_SCK	EVENTOUT	—	—
PB4	SPI1_MISO	TIM3_CH1	EVENTOUT	—
PB5	SPI1_MOSI	TIM3_CH2	TIM16_BKIN	I2C1_SMBA
PB6	USART1_TX	I2C1_SCL	TIM16_CH1N	—
PB7	USART1_RX	I2C1_SDA	TIM17_CH1N	—
PB8	—	I2C1_SCL	TIM16_CH1	—
PB9	IR_OUT	I2C1_SDA	TIM17_CH1	EVENTOUT
PB10	—	I2C1_SCL I2C2_SCL	—	—

续表

引　脚	AF0	AF1	AF2	AF3
PB11	EVENTOUT	I2C1_SDA	—	—
		I2C2_SDA		
PB12	SPI1_NSS	EVENTOUT	TIM1_BKIN	—
	SPI2_NSS			
PB13	SPI1_SCK	—	TIM1_CH1N	—
	SPI2_SCK			
PB14	SPI1_MISO	TIM15_CH1	TIM1_CH2N	—
	SPI2_MISO			
PB15	SPI1_MOSI	TIM15_CH2	TIM1_CH3N	TIM15_CH1N
	SPI2_MOSI			

注：如果使用 STM32F0 的固件库，可在 stm32f0xx_gpio.c 文件中查阅 GPIO_PinAFConfig 函数，ST 公司为便于查找已经在该函数注释中给出相应的对应关系。

5.2　GPIO 固件库

文件 stm32f0xx_gpio.c 包含了 GPIO 固件库函数，主要功能是初始化和配置功能、GPIO 读和写功能、GPIO 复用功能配置。GPIO 函数以及对应功能说明见表 5-5。

表 5-5　GPIO 函数以及对应功能说明

函　数	功 能 说 明
void　GPIO_Init (GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_InitStruct)	根据参数GPIO_InitStruct参数初始化GPIOx外设
void　GPIO_PinAFConfig (GPIO_TypeDef *GPIOx, uint16_t GPIO_PinSource, uint8_t GPIO_AF)	对指定CPIO数据端口写数据
void　GPIO_PinLockConfig (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)	锁定GPIO引脚配置寄存器
uint16_t　GPIO_ReadInputData (GPIO_TypeDef *GPIOx)	读取指定输入端口引脚
uint8_t　GPIO_ReadInputDataBit (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)	读取指定输入端口引脚位
uint16_t　GPIO_ReadOutputData (GPIO_TypeDef *GPIOx)	读取指定GPIO输出数据端口
uint8_t　GPIO_ReadOutputDataBit (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)	读取指定输出数据端口位
void　GPIO_ResetBits (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)	清除选定的数据端口位
void　GPIO_SetBits (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)	将选定端口位置1
void　GPIO_StructInit (GPIO_InitTypeDef *GPIO_InitStruct)	使用默认值填充GPIO_InitStruct
void　GPIO_Write (GPIO_TypeDef *GPIOx, uint16_t PortVal)	对指定端口写数据
void　GPIO_WriteBit (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin, BitAction BitVal)	选定端口置1或清0

使用该驱动函数的过程如下。

（1）使用 RCC_AHBPeriphClockCmd()函数使能 GPIOAHB 时钟。

（2）使用 GPIO_Init()函数配置 GPIO 引脚。每个引脚可能是下面 4 种配置情况。

- 输入：浮空、上拉、下拉。
- 输出：推挽式（上拉、下拉或者无 pull）与开漏模式（上拉、下拉或者无 Pull）。
- 复用功能：推挽式（上拉、下拉或者无 pull）与开漏模式（上拉、下拉或者无 Pull）。
- 模拟：当引脚用作 ADC 通道、DAC 输出或者比较输入。

（3）外设复用功能：

- 对于 ADC、DAC 和比较器情况，使用 GPIO_InitStruct->GPIO_Mode = GPIO_Mode_AN 配置引脚为模拟模式。
- 对于其他外设（TIM、USART）：使用 GPIO_pinAFConfig 函数连接引脚为外设的预期复用功能。PortC、PortD 和 PortF 不必配置；使用 GPIO_InitStruct->GPIO_Mode = GPIO_Mode_AF 配置相应引脚的复用功能模式；通过 GPIO_PuPd、GPIO_OType 和 GPIO_Speed 设置相应的类型、上拉/下拉和输出速度；调用 GPIO_Init()函数。

（4）使用 GPIO_ReadInputDataBit()读取引脚的输入电平。

（5）通过 GPIO_SetBits()/GPIO_ResetBits()对引脚电平置位或者复位。

（6）复位时，复用功能未被激活，引脚浮空（除了 JATG 引脚）。

（7）当 LSE 时钟未用，LSE 时钟引脚 OSC32_IN 和 OSC32_OUT 可被配成 GPIO（PC14 和 PC15），但 LSE 功能比 GPIO 功能优先。

（8）当 HSE 时钟未用，HSE 时钟引脚 OSC_IN/OSC_OUT 可被配成 GPIO（PD0 和 PD1），但 HSE 功能比 GPIO 功能优先。

5.3 GPIO 应用实例

GPIO 的主要用途是作为信号的输入、输出。

GPIO 作为输入功能时主要有开关量（干触点无源输入）或者数字量（TTL 有源输入）。开关量信号（通常叫法干接点）主要是继电器的输出、按钮开关等。干接点在连接时间电压无方向性。数字输入量一般有 DC5V、DC24V、DC48V、AC110V 等电压幅值，接到 GPIO 引脚时，需要根据输入电压值以及端口特性转换成端口允许值（多数 GPIO 端口输入电压是 3.3V，部分端口耐压为 5V）。根据电压幅值转换时输入电源地与 GPIO 引脚对应的电源地是否直接连接在一起，分为隔离输入与非隔离输入两种情况。板载按键是常见的非隔离输入。图 5-3 是按键形式的 IO。PA0.1 对应 STM32F0x 的一个引脚。R10 与 C1 构成滤波器电路实现对按键的去抖。其中 VDD3.3 与 GND 与微控器引脚上的电源（VSS 与 VDD）相同，无隔离。

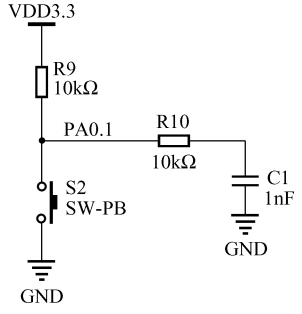


图 5-3　按键输入

光耦和继电器是常见的隔离器件。光耦是以光为媒介传输电信号的器件，将发光器和受光器封装在同一管壳内。当输入端加电信号时发光器发出光线，受光器接收光并产生电流，从而实现电-光-电的转换，达到信号隔离目的。光耦构成的隔离电路具有

耗电少，开关频率高，响应快的优点；但缺点有压降，电流小，易因高压或浪涌损坏。光耦构成的隔离电路适合开关频率高的场合。图 5-4 是光耦隔离输入电路。光耦的输入端 24V 电源以及 24VGND 与微控器的电源无任何关联。PA0.1、PA0.2、PA0.3、PA0.4 是 STM32F0x 微控器相应引脚。

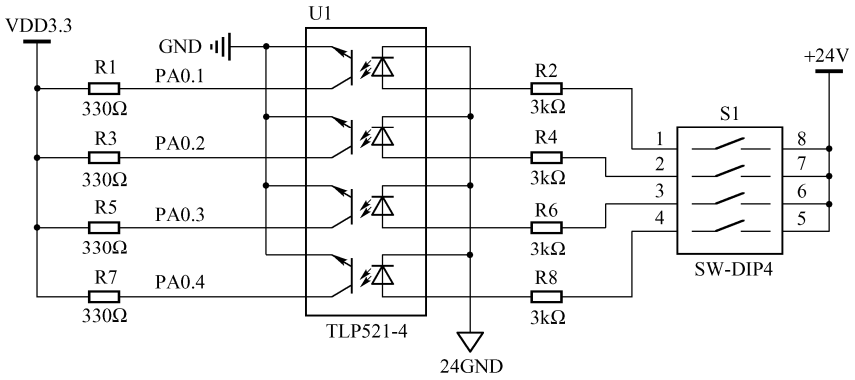


图 5-4　光耦隔离输入电路

继电器是一种当输入量（电、磁、声、光、热）达到一定值时，输出量将发生跳跃式变化的自动控制器件。继电器的优点是无压降、高绝缘、高耐压，不太会出现高压击穿浪涌电流烧毁等问题，可控制大容量负载；缺点是开关频率低，耗电比光耦大。图 5-5 是使用继电器作为接收 I/O 量输入信号的方式。该方式常用于干扰较大的工业环境。图 5-5 中，D1 是续流保护二极管，继电器关断时能量释放，K1 为继电器。

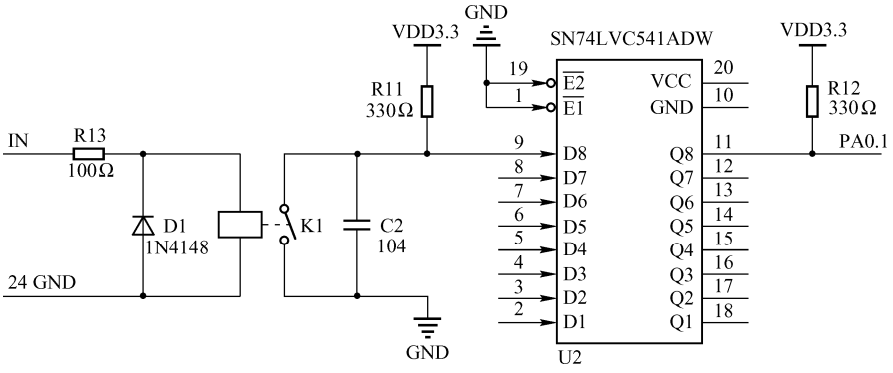


图 5-5　继电器的 I/O 输入

图 5-5 中的 U2（SN74LVC541ADW）是锁存器，用于将继电器的状态锁存，直到继电器状态发生变化，其目的是为了保证微控器能够读到 I/O 状态变化。这是因为在做一般产品设计时，I/O 端口状态的变化很少通过中断接收，只有保护功能才会采用中断接收。不采用中断方式是为了防止程序中断程序过度占用 CPU，造成核心任务与其他中断响应延迟，但降低了 I/O 变化对应任务优先级。

采用隔离与非隔离方式的 I/O 输入对应的代码是相同的。设置 I/O 为输入状态（以 PA0.1 引脚为例）。

```
/* 使能 GPIO 时钟 */
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
/* 配置 PA0 引脚为输入悬浮*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

读引脚状态通过 GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0)。

注意：采用直接读 I/O 状态判断外部输入状态，是建立在外部输入器件带有锁存功能的基础上。或者采用如图 5-5 所示的锁存芯片，或者本身含有锁存功能的自锁按键开关、旋钮之类器件，保证外部状态在被读走之前不丢失。否则只能采用第 6 章要讲解的扩展中断功能。

作为数字量输出时，主要是控制各种类型的门电路、三极管、光耦、蜂鸣器、继电器等。数字量输出的含义是指这种类型的输出信号只有简单的两种状态：高电平和低电平，也可以理解为开（ON）或者关（OFF）两种状态。

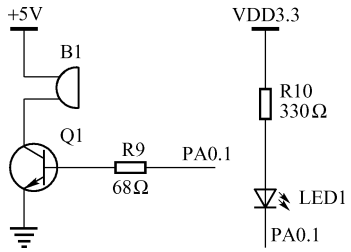


图 5-6　非隔离输出

对于工业现场所需要的数字量信号，具有多种电压等级，这就需要通过不同的输出驱动电路来实现。不同的输出器件可以使数字量输出信号具有不同的输出形式，如晶体管输出、机械继电器输出、固态继电器输出、双向可控硅输出等。根据输出信号与输出电路是否需要共地，分为非隔离或者隔离输出。图 5-6 是常见的非隔离方式输出，用于驱动蜂鸣器以及二极管。

图 5-7 是继电器输出，采用触点输出，工作电流可根据需要选大触点的继电器。优点是工作电流大，但输出时间有一定延迟。

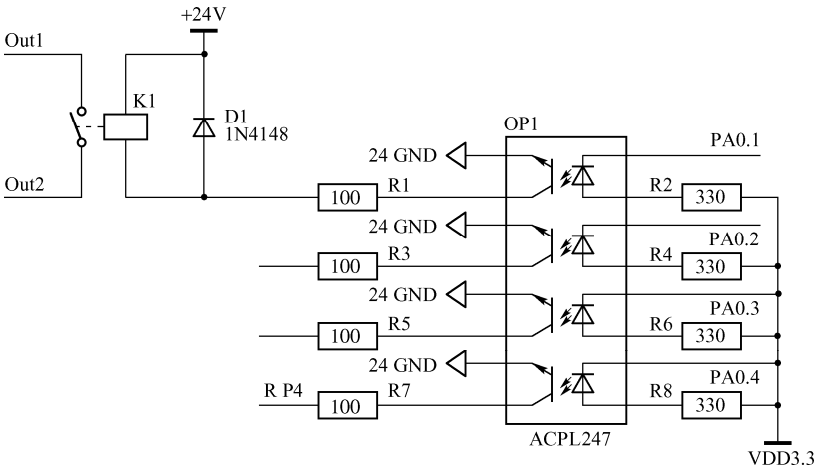


图 5-7　继电器输出

图 5-7 首先采用了光耦隔离，而后驱动继电器的方式。其中光耦输出部分已经 24V。在工作电流较少，开关速度较快场合，可只用光耦部分，不需继电器电路。配合继电器部分主

要用于工作电流较大，同时对开关速度要求不高的场合。

I/O 输出的代码配置如下，以为 PA0.1 为例。

```
/*  使能 GPIO 外设时钟  */
 RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
 /*设置 PA0.1 推挽输出模式*/
 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_ 0;
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
 GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
 GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
 GPIO_Init(GPIOC, &GPIO_InitStructure)
```

输出控制通过表 5-5 中的 GPIO_SetBits()/GPIO_ResetBits()两函数实现。

5.4　小　　结

本章介绍了 STM32F0x 的 GPIO 原理框图、输入/输出情况的配置、复用功能以及固件库内容。最后用实例详细分析了 GPIO 用途，在实际产品设计时如何能够接收不同的 I/O 输入，以及根据需要输出相应的信号，并分析了作为 I/O 输入时的注意事项。

第 6 章

中断和事件

第 1 章提及了 Cortex-M0 的嵌套向量中断控制器，第 4 章提及了中断与超级循环混用是常见的程序结构模式，第 5 章提及了为了快速响应 I/O 引脚状态需要采用中断方式。本章将详细说明 Cortex-M0 的 NVIC 和 STM32F0 的扩展中断原理以及用法。

6.1 嵌套向量中断控制器（NVIC）

NVIC 是一个嵌入式中断控制器，支持低延迟的中断处理，属于 Cortex-M0 的内核外设。处理器内核和 NVIC 紧密结合，并通过寄存器的硬件堆栈以及加载-乘和存储-乘操作的停止和重启来使得中断服务程序（ISR）能够快速执行，极大地缩短了中断延迟。中断处理程序不需要任何汇编封装代码，不用消耗任何 ISR 代码。末尾连锁的优化还极大地降低了从一个 ISR 切换到另一个 ISR 时的开销。

为了优化低功耗设计，将 NVIC 与睡眠模式相结合，提供一个深度睡眠功能，从而使整个器件迅速掉电。

6.1.1 NVIC 概述

嵌套中断向量控制器包含中断管理功能的寄存器，寄存器是以内存映射方式位于系统控制块区域。在 Cortex-M0 处理器中，每个外部中断可独立使能或被禁用，挂起标志可被软件设置或清除。中断源信号可以是电平信号或者触发沿信号，从而使得 NVIC 可用于任何中断源。每个异常有一个优先级，优先级可能是固定的或者可编程的。高优先级的中断可抢占低优先级中断，从而实现中断嵌套。STM32F0 具有 4 个可编程的优先级（使用了 2 位的中断优先级），级别越高对应的优先级越低。因此，级别 0 是最高的中断优先级。

通过 PRIMASK 寄存器，NVIC 提供中断屏蔽功能，可屏蔽除了硬故障和 NMI 以外的所有异常，适合用于对一些实时性强、对任务执行时间要求严格的任务。STM32F0 包含 32 个可屏蔽中断通道（不包含 16 个 Cortex-M0 的中断线）。

表 6-1 是中断管理的 NVIC 寄存器。NVIC 寄存器起始地址是 0xE000E100。

表 6-1　中断管理的 NVIC 寄存器

地　址	名　称	类　型	复 位 值	描　述
0xE000E100	ISER	R/W	0x00000000	中断设置-使能寄存器
0xE000E180	ICER	R/W	0x00000000	中断清除-使能寄存器
0xE000E200	ISPR	R/W	0x00000000	中断设置-挂起寄存器
0xE000E280	ICPR	R/W	0x00000000	中断清除-挂起寄存器
0xE000E400~0xE000E41C	IPR0-7	R/W	0x00000000	中断优先级寄存器

1. 中断使能寄存器与中断清除寄存器

ISER 中断使能寄存器使能中断，并显示哪些中断被使能。ICER 寄存器禁能中断，并显示哪些中断被使能。两个寄存器均为 32 位，以及被支持的中断至多 32 位。被挂起中断被使能，NVIC 根据优先级激活中断。如果中断未被激活，即使中断信号将中断置成挂起状态，但 NVIC 也无法激活中断。

中断使能和清除功能由两个寄存器实现，区别于读-改-写或者第三方寄存器与影子寄存配合的方式。该方式更改寄存器速度快，并可防止在进行读-改-写寄存器过程中被打断。

CMSIS 提供了对使能与禁用寄存器的访问函数如下。

void NVIC_EnableIRQ(IRQn_Type IRQn)：使能一个中断或异常。

void NVIC_DisableIRQ(IRQn_Type IRQn)：禁能一个中断或异常。

2. 中断挂起设置与清除

如果中断没有被立即处理，中断请求将被挂起。挂起标志保存在挂起寄存器，直到中断被处理而清除。通过中断设置挂起（ISPR）和中断清除挂起（ICPR）寄存器可修改或者读取挂起状态。ISPR 强制中断进入挂起状态，并显示哪些中断正在挂起。ICPR 使中断离开挂起状态，并显示哪些中断正在挂起。

通过软件修改中断挂起状态寄存器，可引起中断触发。

void NVIC_SetPendingIRQ(IRQn_Type IRQn)：将中断或异常的挂起状态置位。

void NVIC_ClearPendingIRQ(IRQn_Type IRQn)：将中断或异常的挂起状态清零。

uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)：读取中断或异常的挂起状态。如果挂起状态被设为 1，这个函数就返回非零值。

下面例子演示了通过设置中断使能以及中断挂起方式模拟中断产生的过程。该程序没有配置任何外设，通过使能中断和使能挂起标志进入中断服务程序。观察 EXTI0_1_IRQHandler 函数运行情况，会发现并不运行 if 语句内部的语句，即通过中断设置的软件模拟方式进入中断服务程序，相应外设并无变化。

```
int main(void)
{
  NVIC_EnableIRQ(EXTI0_1_IRQn);
  NVIC_SetPendingIRQ(EXTI0_1_IRQn);
  NVIC_EnableIRQ(WWDG_IRQn);
  NVIC_SetPendingIRQ(WWDG_IRQn);
  /* Infinite loop */
```

```
    while (1)
    {
    }
}
void EXTI0_1_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line0) != RESET)
    {

        EXTI_ClearITPendingBit(EXTI_Line0);
    }
}
        void    WWDG_IRQHandler    (void)
{;
}
```

3. 中断优先级寄存器

IPR0~IPR7 寄存器为每个中断提供了一个两位的优先级域，相比 Cortex-M3 省略了副优先级。这些寄存器只能字访问。每个寄存器包含 4 级优先级，如图 6-1 所示。

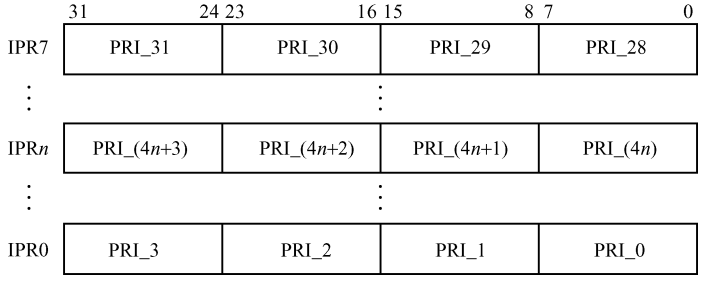


图 6-1　IPR 寄存器

注意：中断优先级在中断使能后不能改变。

CMSIS 中提供了两个优先级操作函数如下。

void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)：设置中断或异常的优先级。

uint32_t NVIC_GetPriority(IRQn_Type IRQn)：读取一个优先级可配置的中断或异常的优先级。

6.1.2　电平中断和脉冲中断

处理器支持电平有效的中断和脉冲中断。脉冲中断也被描述成边沿触发的中断。一个电平有效的中断一直保持有效，直至外设将中断信号撤销。通常，发生这种情况的原因是 ISR 访问外设导致外设将中断请求清除。脉冲中断是在处理器时钟的上升沿同时采样到的一个中断信号。为了确保 NVIC 检测到中断，外设必须使中断信号至少在一个时钟周期内保持有效，在这段时间内 NVIC 检测脉冲并锁存中断。

当处理器进入 ISP 时，它自动消除中断的挂起状态。对于一个电平有效的中断，如果在处理器从 ISR 返回之前中断信号未被撤销，中断再次变成挂起，处理器必须再次执行 ISR。这表示，外设可以一直使中断信号保持有效，直到它不再需要服务为止。

Cortex-M0 锁存所有的中断。外设中断会由于下面的其中一个原因而变为挂起。

- NVIC 检测到中断信号有效，而相应的中断无效；
- NVIC 检测到中断信号的一个上升沿；
- 软件向相应的中断设置-挂起寄存器位写入值。

挂起的中断一直保持挂起，直到出现以下其中一种情况。

- 处理器进入中断的 ISR。这就使中断的状态从挂起变为有效，而且还有如下条件。
  - 对于电平有效的中断，当处理器从 ISR 返回时，NVIC 采样中断信号。如果中断信号有效，中断的状态变回挂起，这可能使得处理器立刻再次进入 ISR。否则，中断的状态变为无效。
  - 对于脉冲中断，NVIC 继续监测中断信号，如果这个中断信号一直处于脉冲状态，中断的状态就变成挂起和有效。在这种情况下，当处理器从 ISR 返回时，中断的状态变为挂起，这可能使得处理器立刻重新进入 ISR。如果当处理器在处理 ISR 时中断信号的脉冲就不存在了，那么，当处理器从 ISR 返回时中断的状态变为无效。
- 利用软件向相应的中断清除-挂起寄存器位写入值。对于电平有效的中断，如果中断信号仍然有效，中断的状态不改变。否则，中断的状态变为无效。对于脉冲中断，中断的状态可变为无效和有效。
  - 无效（如果中断之前的状态是挂起）。
  - 有效（如果中断之前的状态是有效和挂起）。

注意：STM32F0x 仅支持沿中断，不支持电平中断。

6.2　中断和异常向量

表 6-2 是 STM32F51xx 的中断向量表。由于 STM32F0x 的各芯片功能有所不同，带来部分芯片的中断向量有所差异（简单办法是通过 STM32F0 固件库的 stm32f0xx.h 文件中的 IRQn 查询）。比如，表 6-2 中的 ADC_COMP 在 STM32F051 中是 ADC 与比较器 1、2 共享的中断向量，但在 STM32F030X 中缺少比较器功能，仅是 ADC 的中断向量。

表 6-2　STM32F51xx 器件向量表

位置	优先级	优先级类型	名　称	说　明	地　址
	—	—	—	保留（Reserved）	0x0000 0000
	–3	固定	Reset	复位（Reset）	0x0000 0004
	–2	固定	NMI	不可屏蔽中断。RCC时钟安全系统（CSS）连接到NMI向量	0x0000 0008

续表

位置	优先级	优先级类型	名　　称	说　　明	地　　址
	–1	固定	HardFault	所有类型的错误（fault）	0x0000 000C
	3	可设置	SVCall	通用SWI指令调用的系统服务	0x0000 002C
	5	可设置	PendSV	可挂起的系统服务	0x0000 0038
	6	可设置	SysTick	系统嘀嗒定时器	0x0000 003C
0		可设置	WWDG	窗口看门狗中断	0x0000 0040
1		可设置	PVD	连接到EXTI线的可编程电压检测（PVD）中断	0x0000 0044
2		可设置	RTC	RTC全局中断	0x0000 0048
3		可设置	FLASH	Flash全局中断	0x0000 004C
4		可设置	RCC	RCC全局中断	0x0000 0050
5		可设置	EXTI0_1	EXTI线[1:0]中断	0x0000 0054
6		可设置	EXTI2_3	EXTI线[3:2]中断	0x0000 0058
7		可设置	EXTI4_15	EXTI线15和EXTI线4中断	0x0000 005C
8		可设置	TSC	触摸传感中断	0x0000 0060
9		可设置	DMA_CH1	DMA通道1中断	0x0000 0064
10		可设置	DMA_CH2_3	DMA通道2和3中断	0x0000 0068
11		可设置	DMA_CH4_5	DMA通道4和5中断	0x0000 006C
12		可设置	ADC_COMP	ADC和比较器1和2中断	0x0000 0070
13		可设置	TIM1_BRK_UP_TRG_COM	TIM1刹车、更新、触发和通信中断	0x0000 0074
14		可设置	TIM1_CC	TIM1捕获比较中断	0x0000 0078
15		可设置	TIM2	TIM2全局中断	0x0000 007C
16		可设置	TIM3	TIM3全局中断	0x0000 0080
17		可设置	TIM6_DAC	TIM6全局中断和DAC欠载中断	0x0000 0084
18		可设置	Reserved		0x0000 0088
19		可设置	TIM14	TIM14全局中断	0x0000 008C
20		可设置	TIM15	TIM15全局中断	0x0000 0090
21		可设置	TIM16	TIM16全局中断	0x0000 0094
22		可设置	TIM17	TIM17全局中断	0x0000 0098
23		可设置	I2C1	I2C1全局中断	0x0000 009C
24		可设置	I2C2	I2C2全局中断	0x0000 00A0

续表

位置	优先级	优先级类型	名　　称	说　　明	地　　址
25		可设置	SPI1	SPI1全局中断	0x0000 00A4
26		可设置	SPI2	SPI2全局中断	0x0000 00A8
27		可设置	USART1	USART1全局中断	0x0000 00AC
28		可设置	USART2	USART2全局中断	0x0000 00B0
29		可设置	USART3_4	USART3和4全局中断	0x0000 00B4
30		可设置	CEC	CEC、CAN全局中断（连接到外部线27）	0x0000 00B8
31		可设置	USB	USB全局中断（连接到外部线18）	0x0000 00BC

6.3　扩展中断和事件控制器（EXTI）

扩展中断和事件控制器（EXTI）管理外部和内部异步事件/中断，并生成相应给 CPU/中断控制器的事件请求以及给电源管理的唤醒请求。

注：ST 公司针对 EXTI 缩写对应的英文是 Extended interrupts and events controller，有的文档翻译成外部中断和事件控制器，即 Extended 翻译成外部，作者感觉不是很妥。ST 针对 EXTI 的功能描述是：(EXTI) manages the external and internal asynchronous events/interrupts。EXTI 包含了外部和内部中断，即通常意义所说的端口中断（外中断），还包括了 STM32F0 的部分外设中断连接到 EXTI 上。

EXTI 允许管理多达 32 个外部/内部事件线（23 个外部事件线及 9 个内部事件线）。每个扩展中断线可以独立选择触发沿，而内部总为上升沿触发。一个中断可以一直让其挂起。如果发生了扩展中断或事件，状态寄存器指示其中断源；一个事件通常是一个简单的脉冲，用于触发核的唤醒（如 Cortex-M0 的 RXEV 引脚）。对于内部中断，挂起状态由核（IP）产生，所以不需要特定标志。每条输入线可单独屏蔽其产生中断或事件。另外，内部线只有在停止模式下采样。控制器允许软件写特定的寄存器仿真触发相应的线产生事件和中断。

EXTI 主要特性如下：

- ❑ 支持产生多达 32 个事件/中断请求；
- ❑ 作为外部或内部事件请求的每一线都可独立配置；
- ❑ 每个事件/中断线都有独立的屏蔽；
- ❑ 当系统不处于停机（STOP）模式时自动禁止内部线；
- ❑ 独立触发外部事件/中断线；
- ❑ 每个扩展中断线都有专用的状态位；
- ❑ 仿真所有的外部事件请求。

6.3.1　框图

扩展中断/事件框图如图 6-2 所示。

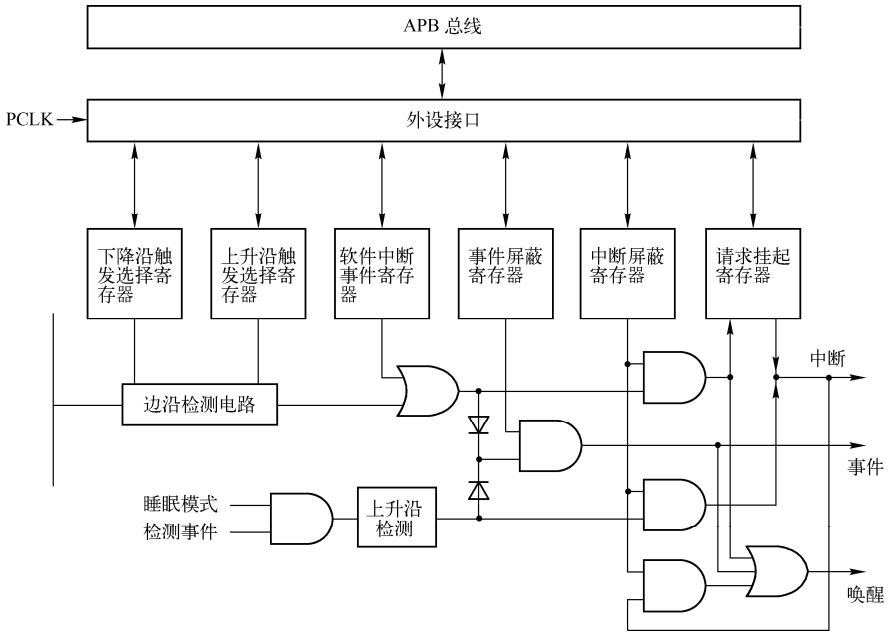


图 6-2　扩展中断/事件框图

6.3.2　事件管理

STM32F0xx 可以处理外部和内部事件来唤醒内核（WFE）。唤醒事件可由下列配置产生。

- 在外设控制寄存器中使能一个扩展中断但不在 NVIC 中使能，同时使能 Cortex-M0 系统控制寄存器中的 SEVONPEND 位。当 MCU 从 WFE 恢复后，需要清除相应外设的中断挂起位和外设 NVIC 中断通道挂起位（在 NVIC 中断清除挂起寄存器中）。
- 配置一个外部或内部 EXTI 线为事件模式，当 CPU 从 WFE 恢复后，不必清除相应外设的中断挂起位或 NVIC 中断通道挂起位。因为对应事件线的挂起位没有被置位。

6.3.3　功能说明

要产生中断，必须先配置好并使能中断线。根据需要的边沿检测设置 2 个触发寄存器，同时在中断屏蔽寄存器的相应位写 1 允许中断请求。当扩展中断线上发生了期待的边沿时，将产生一个中断请求，对应的挂起位也随之被置 1。在挂起寄存器的对应位写 1，将清除该中断请求。

对于内部中断线，触发沿都为上升沿，默认使能，但内部中断线没有相应的挂起位。

如果需要产生事件，必须先配置好并使能事件线。根据需要的边沿检测通过设置 2 个触发寄存器，以及事件屏蔽寄存器置 1。当事件线上发生了选定边沿时，将产生一个事件请求脉冲，对应的挂起位不被置 1。

对于扩展中断线，一个中断/事件请求也可由软件对相应的软件中断/事件寄存器位写 1 来产生。

注：关联到内部线的中断或事件仅在系统处于停止模式下才能被触发。当系统运行时，不会产生该类的中断/事件。

1．硬件中断选择

下列过程可配置 1 条线路为中断。

- 在 EXTI_IMR 寄存器中配置所选中断线的屏蔽位。
- 在 EXTI_RTSR 和 EXTI_FTSR 中配置所选中断线的触发选择位。
- 配置对应到扩展中断控制器（EXTI）的 NVIC 中断通道的使能和屏蔽位，使得中断线中的请求可以被正确地响应。

2．硬件事件选择

根据下列过程可配置 1 条线路作为事件源。

- 在 EXTI_EMR 寄存器中配置所选事件的屏蔽位。
- 在 EXTI_RTSR 和 EXTI_FTSR 中配置所选事件线的触发选择位。

3．软件中断/事件的选择

任何扩展中断线可被配置为软件中断/事件线。以下过程产生一个软件中断。

- 在 EXTI_IMR、EXTI_EMR 中配置相应的屏蔽位。
- 在软件中断寄存器（EXTI_SWIER）设置相应的请求位。

6.3.4　外部和内部中断/事件线映像

共有最多 28 条中断/事件线可用：7 条线为内部（包含 1 条保留）的和 21 条线为外部的。GPIO 连接到 16 个扩展中断/事件线的方式如图 6-3 所示。

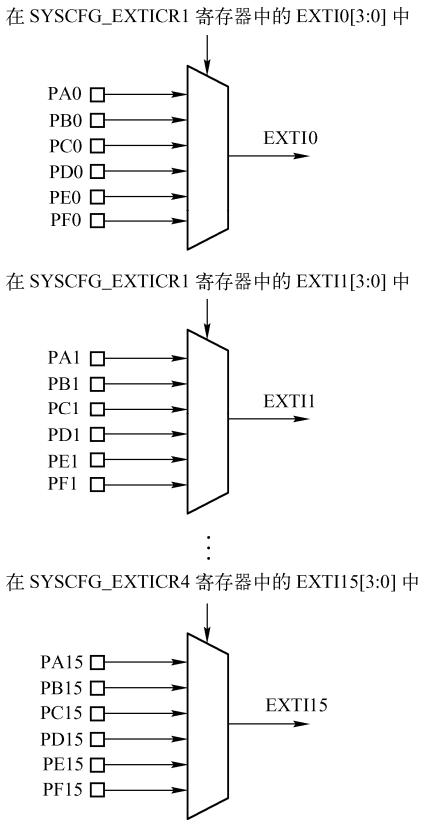


图 6-3　扩展中断/事件 GPIO 映像

EXTI16~EXTI31（除部分保留以及内部线）被连接到其他外设输出上。

6.4 EXTI 固件库

EXTI 的寄存器分为屏蔽寄存器、沿触发寄存器、软件中断以及挂起中断类型的寄存器。屏蔽寄存器有两个中断屏蔽寄存器（EXTI_IMR）与事件屏蔽寄存器（EXTI_EMR），对相应位置 1 是开放来自线 x 上的中断（或事件）请求；写 0 是屏蔽来自线 x 上的中断（或事件）请求。沿触发寄存器分别是上升沿触发选择寄存器（EXTI_RTSR）与下降沿触发选择寄存器（EXTI_FTSR），其中写 0 为禁用，写 1 为使能。将软件中断事件寄存器（EXTI_SWIER）写 1 将使相应的 EXTI 线产生中断。EXTI 的库函数方便了读/写这些寄存器，见表 6-3。

表 6-3 EXTI 固件库函数

函 数 名	描 述
EXTI_DeInit	将外设EXTI寄存器重设为默认值
EXTI_Init	根据EXTI_InitStruct中指定的参数初始化外设EXTI寄存器
EXTI_StructInit	把EXTI_InitStruct中的每一个参数按默认值填入
EXTI_GenerateSWInterrupt	产生一个软件中断
EXTI_GetFlagStatus	检查指定的EXTI线路标志位设置与否
EXTI_ClearFlag	清除EXTI线路挂起标志位
EXTI_GetITStatus	检查指定的EXTI线路触发请求发生与否
EXTI_ClearITPendingBit	清除EXTI线路挂起位

设置一个 I/O 为扩展中断源的过程如下。

① 使用 GPIO_Init()配置 I/O 引脚为输入模式。

② 使用 SYSCFG_EXTILineConfig()函数将输入源引脚为 EXTI 线。

③ 使用 EXTI_Init()函数设置模式（中断、事件）和沿触发方式（上升沿、下降沿或二者）。对于内部中断，触发沿不需配置（固定的是上升沿）。

④ 使用 NVIC_Init()配置 EXTI 线的 NVIC IRQ 通道。

⑤ 可使用 EXTI_GenerateSWInterrupt()产生软件中断。

⑥ 使用 RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE)打开 SYSCFG_EXTICRx 寄存器的写保护。

6.5 EXTI 中断实例

STM32F0 的 EXTI 中断没有电平中断功能，在做产品设计时，需要避开该需求。STM32F0 支持下降沿中断、上升沿中断或者上升下降同时存在情况。

下面代码是通过 PA0.1 端口模拟沿触发的过程。图 6-3 显示了 PA0.1 是连接到 EXTI1 上，所以对应的中断向量是 EXTI0_1_IRQHandler，在 EXTI0_Config 函数中通过 NVIC_InitStructure.NVIC_IRQChannel = EXTI0_1_IRQn 设置。在 EXTI0_Config 函数配置 Exti1 线

中断。首先设置 PA0.1 引脚为输入引脚。通过 SYSCFG_EXTILineConfig()函数将输入源引脚为 EXTI 线。通过 EXTI_Init 函数设置上升沿触发中断。

由于 EXTI 中断的触发源可以多个，所以在 EXTI0_1_IRQHandler 中需要首先通过 EXTI_GetITStatus 判断中断源。

EXTI0_1_IRQHandler 中的 EXTI_ClearITPendingBit(EXTI_Line1)是清除 EXTI 的挂起中断标志，需要注意的是在后续章节的中断服务程序中，有的程序显性调用命令清除中断挂起标志，有的无显性清除指令。这是由于部分外设（例如 USART 外设）进入中断服务程序后，读寄存器的过程自动清除中断挂起标志。

程序的执行可以通过 PA0.1 引脚触发，也可通过 EXTI_GenerateSWInterrupt(EXTI_Line1)模拟。

```
static void EXTI0_Config(void)
{
  /* 使能 GPIOA 时钟 */
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
  /* 配置 PA0.1 输入引脚 */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  /* 使能 SYSCFG 时钟 */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
  /* 连接 EXTI0 线到 PA1 引脚 */
  SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource1);

  /* 配置 EXTI0 线 */
  EXTI_InitStructure.EXTI_Line = EXTI_Line1;
  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStructure);

  /* 使能、设置 EXTI0 中断 */
  NVIC_InitStructure.NVIC_IRQChannel = EXTI0_1_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPriority = 0x01;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
void EXTI0_1_IRQHandler(void)
{
  if(EXTI_GetITStatus(EXTI_Line1) != RESET)//判断来自 EXTI1
  {
      LED_Toggle();
  EXTI_ClearITPendingBit(EXTI_Line1);//清除 EXTI1 线上的挂起标志
  }
```

```
}
```

6.6 HardFault 异常调试实例

HardFault 异常可能是非法的指令码、非法的总线或存储器访问（地址非法），或者类似试图非法切换到 ARM 状态引起的。其中编程错误可能原因是非执行区执行代码、切换 ARM 状态（由于 Cortex-M0 支持 Thumb 指令）、非对齐访问、非法执行 SVC、异常的非法 EXC_RETURN 值均会引起 HardFault 异常。

通过 MDK（版本 4.7）可以跟踪引起 HardFault 异常的语句。为了演示 MDK 跟踪过程，代码中包含了两次 HardFault。

```
struct time
 {
    int hour;
 };

int main(void)
{
 struct   time *p=(struct time   *) 0;
 *((char *) main) = 0;
       p->hour =0;
 while(1)
 {;}
}
```

HardFault 的中断处理程序 HardFault_Handler()比较简单，只有一个 while(1)语句。

通过调试方式运行本程序后，选择 Debug 菜单中的 Stop 停止程序，会发现断点停在了 HardFault_Handler()中，而不是 main 函数的 while 循环处。打开 View 菜单中的 Call Stack Window 页面，如图 6-4 所示。

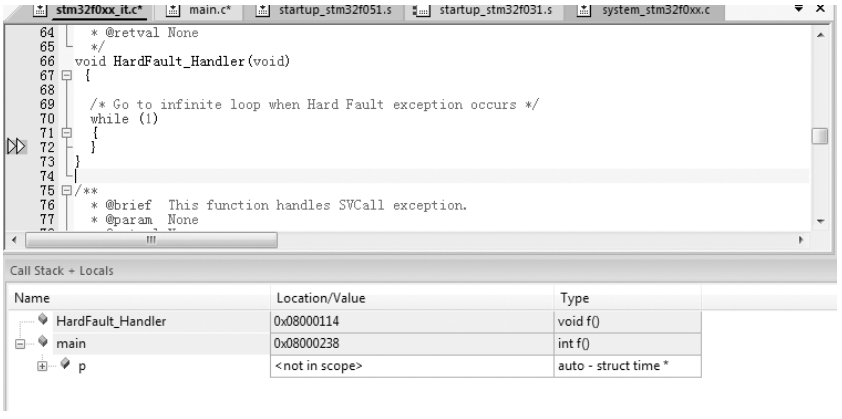


图 6-4　堆栈调用窗口

在 Call-Stack 界面中的 HardFault_Handler 处右击鼠标（见图 6-5），选择 Show Caller Code，会出现图 6-6 所示提示，即由*((char *) main) = 0 语句引起的 HardFault。

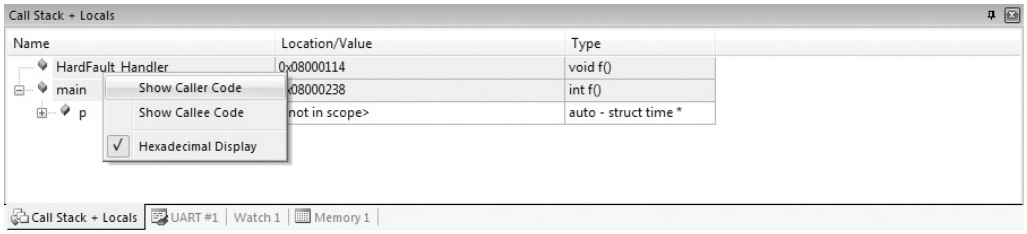


图 6-5　显示调用者

```
61
62  int main(void)
63  {
64    struct  time *p=(struct time  *) 0;
65  *((char *) main) = 0;
66    p->hour =0; //
67    while(1)
68    {;}
69  }
70
71
```

图 6-6　故障源

Joseph Yiu 的《The Definitive Guide to the ARM Cortex-M0》一书关于 HardFault 有更详尽的描述，有兴趣的读者可以参考。

6.7　小　　结

本章讲解了 Cortex-M0 的嵌套中断管理器，以及 STM32F0x 的 EXTI 中断，并给出了如何使用软件模拟中断，EXTI 中断的使用例程，以及在开发产品过程经常遇到的 HardFault 故障的解决方案。嵌套中断为了达到抢占目的，使得嵌套中断程序相比非嵌套中断程序复杂得多，首先是优先级的确定，这需要根据产品特点、产品安全保护要求进行设定，无章法可言。另外嵌套中断在软件编程时需要注意函数重入，高优先级的中断服务程序尽可能设计简短，保证快速退出不影响低优先级中断逻辑。在做产品设计时，需要注意 STM32F0x 的 EXTI 中断不支持电平中断。

第 7 章

通用同步异步收发器（USART）

STM32F0x 的通用同步异步收发器（USART）可与外部设备通过工业标准 NRZ 的形式实现全双工异步串行数据通信。USART 使用分数波特率发生器，从而提供了超宽的波特率范围。USART 支持同步通信模式、半双工单线通信、LIN（本地互联网络）、智能卡协议、IrDA（红外数据协会）SIR ENDEC 规范和 MODEM 流控操作（CTS/RTS），同时还支持多机通信方式。可以使用 DMA 实现多缓冲区设置，从而能够支持高速数据通信。

STM32F0x 的 USART 相比 STM32F1/2 的 USART 主要增加的功能有用于 RS-485 的输出使能（方向控制），发送、接收引脚可软件交换，对 Modbus 协议的基本支持，接收超时检测，数据块结尾中断，地址/字符匹配中断，自动波特率检测，数据位序可配置（MSB 或 LSB）。

USART 区别于 SPI 与 I2C 收发引脚直连的形式，USART 相互通信收发引脚是交叉相连形式。如果不小心将 USART 引脚设计成直连形式，STM32F0x 的发送、接收引脚的软件交换功能就大有作为。

7.1 USART 主要功能

STM32F0x 的 USART 主要功能如下：

- 全双工，异步通信。
- NRZ 标准格式（mark/space）。
- 可配置的 16 倍或 8 倍过采样方法，提供了速度和时钟容忍度间的灵活选择。
- 小数波特率发生器，波特率高达 6Mbit/s（时钟频率为 48MHz，过采样率为 8 倍时）。
- 双时钟驱动支持。
- UART 从 Stop 模式唤醒功能，方便波特率改变，不依赖对 PCLK 的改变。
- 自动波特率检测。
- 数据字长可编程（8 或 9 位）。
- 高位在前或低位在前可设置。
- 停止位个数可设置支持 1 个或 2 个停止位。
- 同步模式下时钟输出功能，实现同步通信。
- 单线半双工通信。

- DMA（直接内存访问）支持下的连续数据通信，利用 DMA 功能将收/发字节缓冲到保留的 SRAM 空间。
- 针对接收器和发送器的单独的使能位。
- 针对发送和接受的单独的信号极性控制。
- 可配置为 Tx/Rx 引脚互换。
- 用于 MODEM 的硬件流控制和 RS-485 发送使能控制。
- 发送检测标志有接收缓冲区满、发送缓冲区空、忙和发送结束标志。
- 校验控制：发送奇偶校验位、接收数据的奇偶检查。
- 4 种错误检测：溢出错误、噪声检测错误、帧错误、校验错误。
- 14 个中断源和中断标志：CTS 切换、LIN 断开检测、发送数据寄存器空、发送完成、接收数据寄存器满、检测到线路空闲、溢出错误、帧错误、噪声错误、奇偶错误、地址/字符匹配、接收超时中断、块结束中断、从 Stop 模式唤醒。
- 多机通信：如果地址不匹配则进入静默模式。
- 从静默模式唤醒（检测到线路空闲或检测到地址标记）。
- 两个接收唤醒模式：地址位（MSB，第 9 位），线路空闲。
- LIN 主机的断开信号发送能力和 LIN 从机的断开信号检测能力：将 USART 硬件设置成 LIN 模式时，有 13 位的断开信号发生器和 10/11 位的断开信号检测功能。
- IrDA SIR 编解码器，普通模式下支持 3/16 位时长。
- 智能卡模式：支持 ISO/IEC7816-3 标准定义的 T=0 和 T=1 智能卡异步协议；智能卡用到 1.5 个停止位。
- 支持 ModBus 通信：超时检测功能；CR/LF 字符识别。

7.2　STM32F0x 的 USART 功能实现

表 7-1 是 STM32F0 系列芯片的功能实现。其中 X 表示被支持，NA 表示不被支持。

表 7-1　STM32F0x 的 USART 功能

USART 模式特征	STM32F03x	STM32F05x		STM32F04x		STM32F07x	
	USART1	USART1	USART2	USART1	USART2	USART1/USART2	USART3/USART4
针对调制解调器的硬件流控制	X	X	X	X	X	X	X
通过 DMA 的连续通信	X	X	X	X	X	X	X
多处理器通信	X	X	X	X	X	X	X
同步模式	X	X	X	X	X	X	X
智能卡模式	X	X	NA	X	NA	X	NA
线半双工通信	X	X	X	X	X	X	X
IrDA SIR ENDEC	X	X	NA	X	NA	X	NA

续表

USART 模式特征	STM32F03x	STM32F05x		STM32F04x		STM32F07x	
	USART1	USART1	USART2	USART1	USART2	USART1/USART2	USART3/USART4
LIN 模式	X	X	NA	X	NA	X	NA
双时钟域和停止模式唤醒	X	X	NA	X	NA	X	NA
接收器超时中断	X	X	NA	X	NA	X	NA
Modbus 通信	X	X	NA	X	NA	X	NA
自动波特率检测	2（模式 0/1）	2（模式 0/1）	NA	4	NA	4	NA
RS-485 用的驱动使能信号	X	X	X	X	X	X	X
USART 数据长度	8、9 位			7、8、9 位			

7.3 USART 功能描述

7.3.1 USART 框图

图 7-1 显示了 USART 双向通信最少需有两个引脚：接收数据输入（RX）和发送数据输出（TX）。这也是比较普遍的用法，比如 RS-232 与 RS-485 接口，均使用 RX 和 TX 引脚，外加一个串行通信电平转换芯片实现。

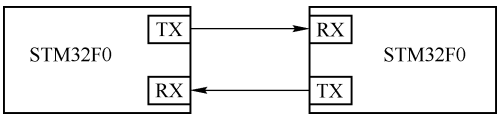


图 7-1　USART 通信引脚连接图

- RX：接收数据输入，是串行数据的输入口。使用过采样技术来完成数据恢复，以区别输入数据和噪声。
- TX：数据发送输出。当发送器被禁止，输出脚回到其 I/O 口配置状态。当发送器被使能，但不发送数据时，TX 脚为高电平输出。在单线和智能卡模式中，这个口线既用于发送数据也用于接收数据。

通过这些引脚，串行数据用数据帧的形式发送和接收。

图 7-2 是 STM32F030x4/x6/x8 的 UART 原理框图，与 STM32F0x1/x2/x8 的原理框图稍微不同，增加了部分功能。SCLK（时钟输出）引脚在同步模式和智能卡模式中被用到，该引脚会输出发送数据的同步时钟，发送功能和 SPI 主模式一致（起始位和停止位上没有时钟脉冲，在最后一位数据上可选择有无时钟脉冲）。同时，数据可经由 RX 引脚同步接收，这可以被用来连接那种通过移位寄存器相连的外设（例如：LCD 驱动器）。时钟相位和极性可软件设定。智能卡模式中，SCLK 引脚会向智能卡提供时钟。nCTS 和 nRTS 用于支持硬件流

控制模式。

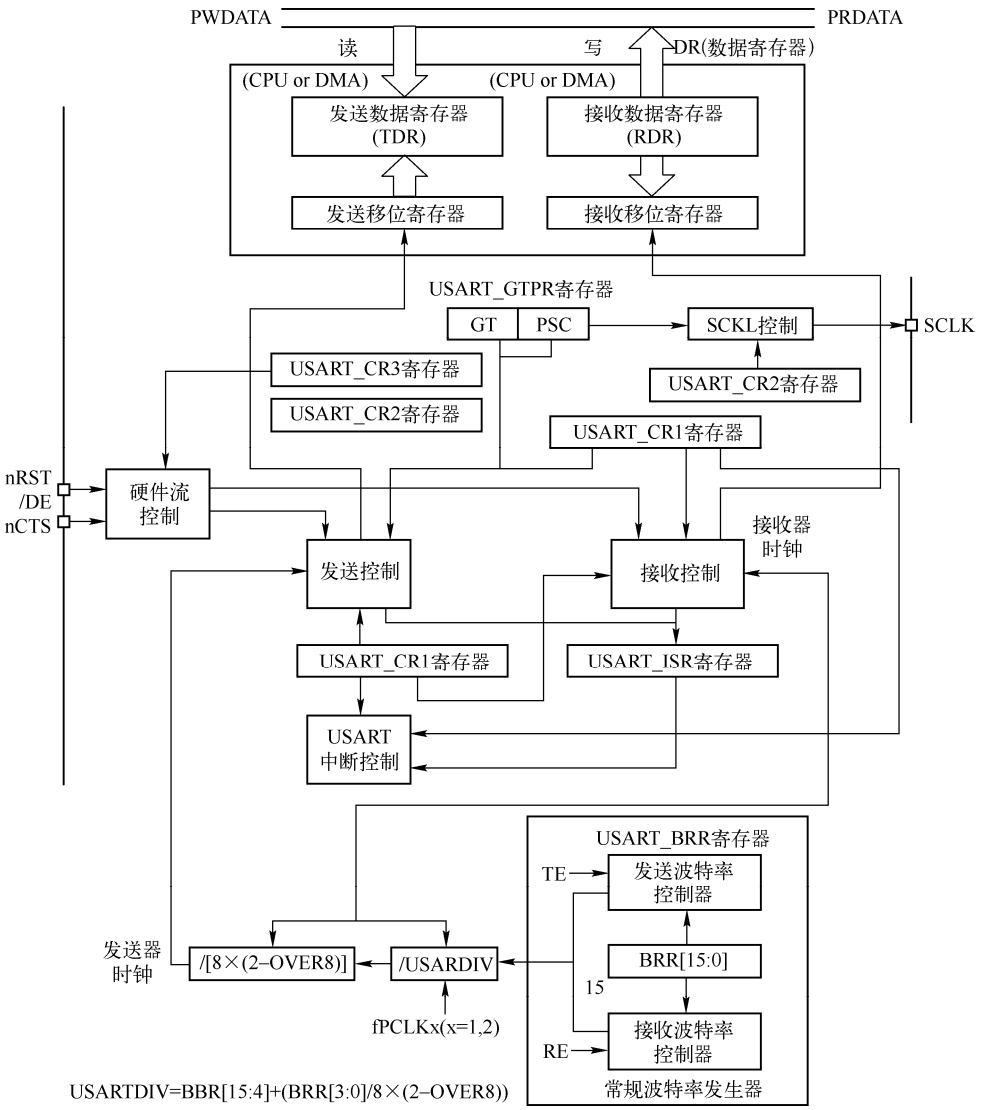


图 7-2　STM32F030x4/x6/x8 的 UART 原理框图

- nCTS：低电平发送，当高电平时作为发送阻塞信号。
- nRTS：低电平代表请求发送，表明 USART 已经准备好接收数据。

DE 引脚用于 RS-485 驱动使能，将外部收发器的发送模式激活。

注：DE 和 nRTS 共用同一个外部引脚。

7.3.2　USART 字符描述

配置 USART_CR1 寄存器中的 M 位选择 8 位或 9 位字长（见图 7-3）。默认设置中，发送和接收的起始位都是低电平，而停止位都是高电平。这个逻辑可以在极性控制中单独地设置

为反向。空闲符号被视为完全由“1”组成的完整的数据帧，后面跟着包含了数据的下一帧的开始位（“1”的位数也包括了停止位的位数）。

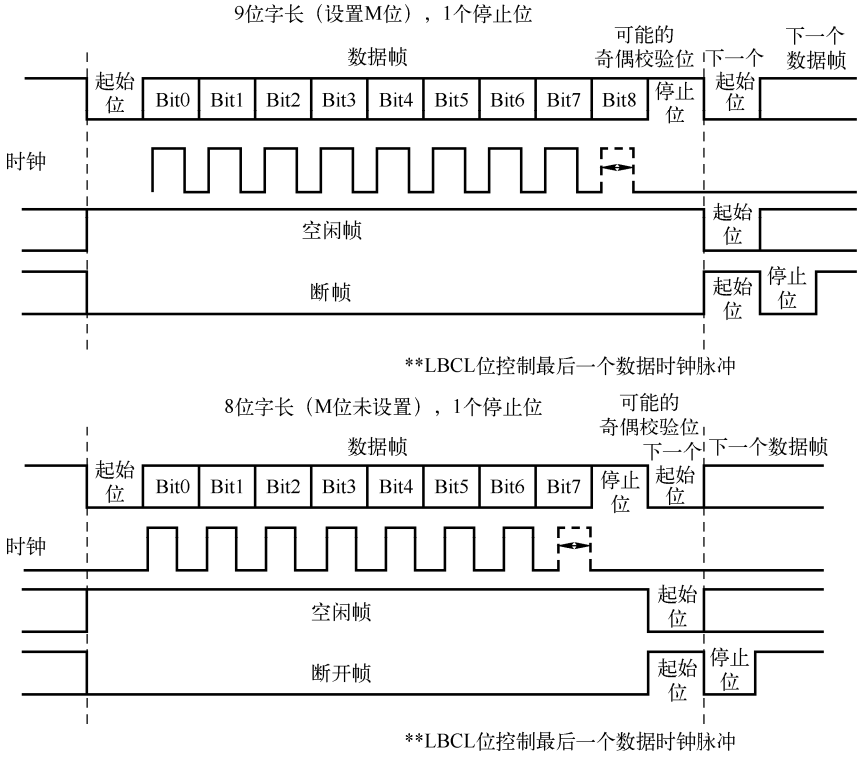


图 7-3　字长设置

断开符号被视为在一个帧周期内全部收到“0”（包括停止位期间，也是“0”）。在断开帧结束时，发送器会再插入 2 个停止位。发送和接收由一个共用的波特率发生器驱动，当发送器和接收器的使能位分别置 1 时，分别为其产生时钟。

7.3.3　发送器

发送器根据 M 位的状态发送 8 位或 9 位的数据字。发送使能位必须置 1 以打开发送功能。发送移位寄存器中的数据在 TX 脚上输出，相应的时钟脉冲在 CK 脚上输出。

1．字符发送

在 USART 发送期间，在 TX 引脚上首先移出数据的最低有效位。在此模式里，USART_TDR 寄存器充当了一个内部总线和发送移位寄存器之间的缓冲器（TDR）。每个字符之前都有一个低电平的起始位之后跟着停止位，停止位的数目是可选择的。USART 支持多种停止位的选择：0.5、1、1.5 和 2 个停止位。

注：（1）在向 USART_TDR 写数据之前必须先令 TE 位为 1。

（2）在 TE 位被置 1 后将会发送一个空闲帧。

2．可配置的停止位

随每个字符发送的停止位的位数可以通过控制寄存器 2 的位 13、12 进行编程，如图 7-4

所示。

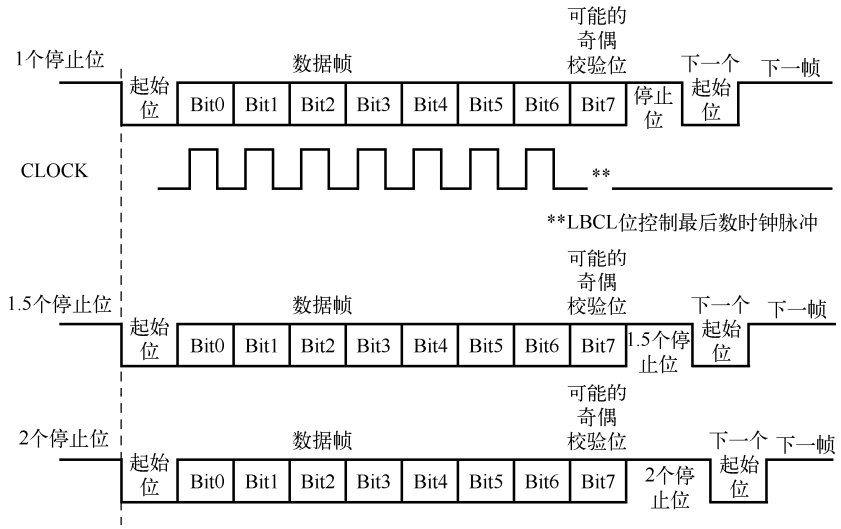


图 7-4　可配置的停止位

（1）1 个停止位：停止位的位数默认。

（2）2 个停止位：可用于常规 USART 模式、单线模式以及调制解调器模式。

（3）1.5 个停止位：在智能卡模式下发送和接收数据时使用。

空闲帧包括了停止位。断开帧是 10 位低电平（当 m=0 时）；或者 11 位低电平（m=1 时），后跟 2 个停止位。不可能传输更长的断开帧（长度大于 10 或者 11 位的那种）。

3．发送配置步骤

（1）设置 USART_CR1 的 M 位来定义字长。

（2）利用 USART_BRR 寄存器选择希望的波特率。

（3）在 USART_CR2 中设置停止位的位数。

（4）将 USART_CR1 寄存器的 UE 位置 1 来使能 USART。

（5）如果采用多缓冲器通信，配置 USART_CR3 中的 DMA 使能位（DMAT）。按多缓冲器通信中的描述配置 DMA 寄存器。

（6）设置 USART_CR1 中的 TE 位，发送一个空闲帧作为第一次数据发送。

（7）把要发送的数据写进 USART_TDR 寄存器（此动作将清除 TXE 位）。在只有一个缓冲器的情况下，对每个待发送的数据重复步骤（7）。重复步骤（7）之前应等待 TXE 变成 1。

（8）在 USART_TDR 寄存器中写入最后一个数据字后，要等待 TC=1，它表示最后一个数据帧的传输结束。当需要关闭 USART 或需要进入停机模式之前，需要确认传输结束，避免破坏最后一次传输。

4．单字节通信

清零 TXE 位总是通过对数据寄存器的写操作来完成的。TXE 位由硬件来设置，如图 7-5 所示。它表明：

（1）数据已经从 TDR 移送到移位寄存器，数据发送已经开始。

（2）USART_TDR 寄存器被清空。

（3）下一个数据可以被写进 USART_TDR 寄存器而不会覆盖先前的数据。

如果 TXEIE 位被设置，该事件将产生一个中断请求。如果此时 USART 正在发送数据，对 USART_TDR 寄存器的写操作将把数据存进 TDR 寄存器，并在当前传输结束时把该数据复制进移位寄存器。如果 USART 没有在发送数据，对 USART_TDR 寄存器的写操作将导致直接把数据放进移位寄存器，数据传输开始，TXE 位则立即被置起。

当一个字节发送完成时（停止位发送后）并且 TXE 被置位时，TC 位会被置 1。如果 USART_CR1 寄存器中的 TCIE 位是 1 时，则会产生中断。

在 USART_TDR 寄存器中写入了最后一个数据字后，在关闭 USART 模块之前或设置微控制器进入低功耗模式之前，必须先等待 TC=1。（见图 7-5：发送时 TC/TXE 的变化情况）

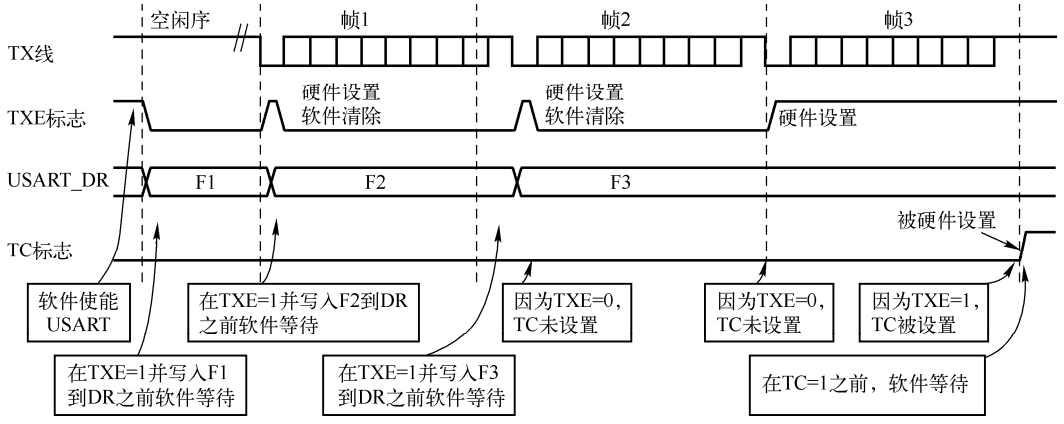


图 7-5　发送时 TC/TXE 的变化情况

注：关闭 USART 的正确步骤如下。

（1）清除 TE 位（如果有数据正在传输或者 USART_寄存器中还有待发送的数据，会在关闭动作生效前发送完）。

（2）TC 位会在发送完毕后被置 1。

（3）在 TC=1 后清除 UE 位。

5．断开符号

向 SBKRQ 位写 1 可发送一个断开符号。断开符号的长度取决于 M 位。如果设置 SBK=1，在完成当前数据发送后，将在 TX 线上发送一个断开符号。SBKF 位会在写操作发生时置 1，在断开符号发送完成时（在断开符号的停止位发出时）被硬件清零。USART 在最后一个断开帧的结束处插入一个逻辑“1”并保持 2 位时长作为停止，以保证能识别下一帧的起始位。

6．空闲符号

将 TE 置 1 将使得 USART 在第一个数据前发送一空闲符号。

7.3.4　接收器

接收器根据 USART_CR1 寄存器中 M 位的状态接收 8 位或 9 位的数据字。

1．起始位侦测

过采样率设置成 16 或 8，不影响起始位侦测的顺序，如图 7-6 所示。在 USART 中，如果辨认出一个特殊的采样序列，那么就认为侦测到一个起始位。该序列为 1110X0X0X0X0X0X0。

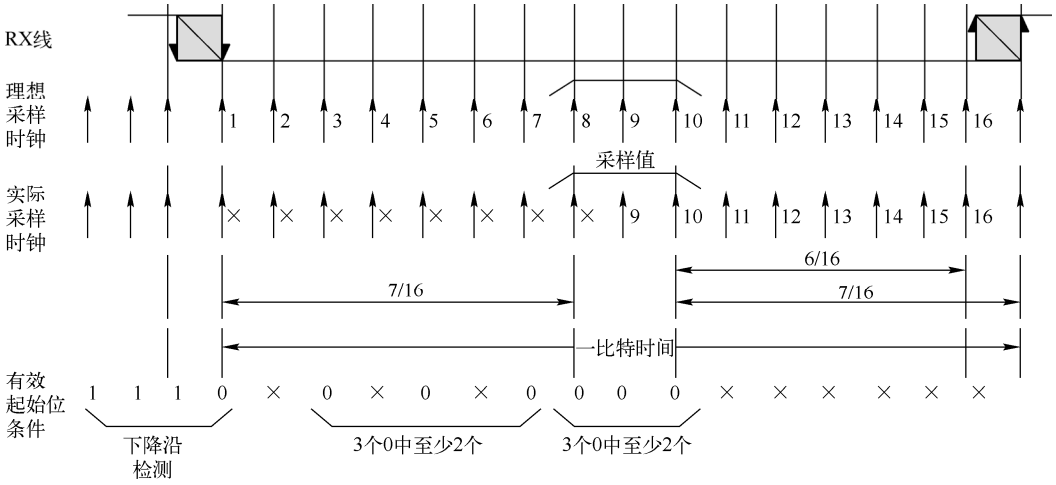


图 7-6　以 16 或 8 过采样时的起始位检测

注：如果该序列不完整，那么接收端将退出起始位侦测并回到空闲状态（不设置标志位）开始等待下降沿。

如果 3 个采样点都为“0”（在第 3、5、7 位的第一次采样和在第 8、9、10 位的第二次采样都为“0”），则确认收到起始位，这时 RXNE 标志会由硬件置 1。如果这时 RXNEIE=1，则会产生中断请求。

如果两次 3 个采样点上仅有 2 个是“0”（第 3、5、7 位的采样点和第 8、9、10 位的采样点），那么起始位仍然是有效的，但是会设置 NE 噪声标志位。如果不能满足这个条件，则中止起始位的侦测过程，接收器会回到空闲状态（不设置标志位）。

如果有一次 3 个采样点上仅有 2 个是“0”（第 3、5、7 位的采样点或第 8、9、10 位的采样点），那么起始位仍然是有效的，但是会设置 NE 噪声标志位。

如果 3 个采样点都为“0”（在第 3、5、7 位的第一次采样和在第 8、9、10 位的第二次采样都为“0”），则确认收到起始位，这时 RXNE 标志会由硬件置 1。如果这时 RXNEIE=1，则会产生中断请求。

如果两次 3 个采样点上仅有 2 个是“0”（第 3、5、7 位的采样点和第 8、9、10 位的采样点），那么起始位仍然是有效的，但是会设置 NE 噪声标志位。如果不能满足这个条件，则中止起始位的侦测过程，接收器会回到空闲状态（不设置标志位）。

如果有一次 3 个采样点上仅有 2 个是“0”（第 3、5、7 位的采样点或第 8、9、10 位的采样点），那么起始位仍然是有效的，但是会设置 NE 噪声标志位。

2．字符接收

在 USART 接收期间，数据的最低有效位（默认情况下）首先从 RX 脚移进。在此模式里，USART_RDR 寄存器充当了一个位于内部总线和接收移位寄存器之间的缓冲器。配置步

骤如下。

（1）设置 USART_CR1 的 M 位来定义字长。

（2）利用波特率寄存器 USART_BRR 选择希望的波特率。

（3）在 USART_CR2 中设置停止位的位数。

（4）通过将 USART_CR1 寄存器的 UE 位置 1 来激活 USART。

（5）如果采用多缓冲器通信，配置 USART_CR3 中的 DMA 使能位（DMAR）。按多缓冲器通信中的描述配置 DMA 寄存器。

（6）将 USART_CR1 的 RE 位置 1。这将激活接收器，使它开始寻找起始位。

当一个字符被接收到时，具有如下特征。

- RXNE 位被置 1。它表明移位寄存器的内容被转移到 RDR。换句话说，数据已经被接收并且可以被读出（包括与之有关的错误标志）。
- 如果 RXNEIE 位是 1，将会引起中断请求。
- 在接收期间如果检测到帧错误、噪音或溢出错误，错误标志将被置起。PE 标志也会和 RXNE 一起被置 1。
- 在多缓冲器通信时，RXNE 在每个字节接收后被置起，并由 DMA 对数据寄存器的读操作而清零。
- 在单缓冲器模式里，由软件读 USART_RDR 寄存器完成对 RXNE 位清除。RXNE 标志也可以通过对 USART_RQR 寄存器中的 RXFRQ 位写 1 来清除。RXNE 位必须在下一字符接收结束前被清零，以避免溢出错误。

3．断开符号

当接收到一个断开帧时，USART 会像处理帧错误一样处理它。

4．空闲符号

当一空闲帧被检测到时，其处理步骤和接收到普通数据帧一样，但如果 IDLEIE 位为 1 将产生一个中断请求。

5．溢出错误

如果 RXNE 还没有被复位，而又接收到了一个字符，则发生溢出错误。数据只有当 RXNE 位被清零后才能从移位寄存器转移到 RDR 寄存器。RXNE 标记是接收到每个字节后被置 1 的。如果下一个数据已被收到或先前 DMA 请求还没被服务，此时 RXNE 标志仍是置位的，也会产生溢出错误。当出现溢出错误时，具有如下特征。

- ORE 位被置 1。
- RDR 内容将不会丢失。读 USART_RDR 寄存器仍能得到先前的数据。
- 移位寄存器中以前的内容将被覆盖。随后接收到的数据都将丢失。
- 如果 RXNEIE 位被置为 1 或 EIE 位被置为 1，会产生中断。
- ORE 位可以通过将 ICR 寄存器中的 ORECF 位置 1 的方式清除。当 ORE 位置 1 时，表明至少有 1 个数据已经丢失。有两种可能性：
  - 如果 RXNE=1，尚一个有效数据还在接收寄存器 RDR 上，可以被读出。
  - 如果 RXNE=0，这意味着上一个有效数据已经被读走，RDR 已经没有内容可读。当上一个有效数据在 RDR 中被读取的同时又接收到新的（也就是丢失的）数据时，此种情况是可能发生的。

6．选择时钟源和适当的过采样率的方法

UART 的时钟源的选择要通过时钟控制系统，必须在使能 USART 之前（将 UE 位置 1），从而打开 USART 的时钟源。

时钟源的选择需要依据两个标准。

- 在低功耗模式下使用串口的可能性。
- 通信速度。

时钟源的频率是 f_{CK}。时钟源如图 7-7 所示。当使用双时钟域和从 Stop 模式唤醒的功能时，时钟源可以是配置成图中的任意一个，默认是 PCLK。

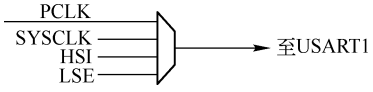


图 7-7　UART 的时钟源

选择 LSE、HSI 作为时钟源，USART 在 MCU 处于低功耗状态的时候还能接收数据。基于唤醒模式选择和接收到的数据，USART 会将 MCU 唤醒，并由软件或者 DMA 将收到的数据从 USART_RDR 寄存器中读走。

如果用其他的时钟源，必须先将其打开以保证 USART 通信。通信速度范围（特别是最大通信速度）是由所选的时钟源来决定的。接收器根据用户设定的过采样率来执行数据恢复，从而将接收数据和噪声区分开来。如果设成同步模式则不会是这样。这需要在最大通信速度和噪声抗扰度及对波特率偏差的敏感度之间做取舍。

通过 USART_CR1 寄存器中的 OVER8 位来选择过采样率是波特率时钟的 8 倍或者是 16 倍。根据应用有两种选择。

- 选择 8 倍过采样以实现较高速度（高至 $f_{CK}/8$）。这种情况下最大的接收器允许的波特率偏差要小一些。
- 选择 16 倍过采样率（OVER8=0）可提高时钟偏差容忍度。这种情况下，最高通信速度就会被限制在 $f_{CK}/16$。

USART_CR3 中的 ONEBIT 位用来选择判断逻辑电平的方法，有两种选择。

- 根据接收数据位中央的三个采样结果进行多数票决。这种情况下，当发现三个参与票决的采样结果不等时，NF 位会被置 1。
- 根据接收数据位中央的单个置采样结果来直接裁决。
  - 在噪声干扰环境中应该选择三个采样多数票决的方式，可以排除检测到噪声的数据，因为那说明在数据采样的时候有干扰。
  - 在不担心噪声的情况下，应选择单采样裁决（ONEBIT=1），可以提高接收器对时钟偏差的容忍度。这种情况下 NF 位永远都不会置 1。

当一帧数据中检测到噪声时，具有如下特征。

- 在 RXNE 位的上升沿，NF 位会被置 1。
- 有问题的数据仍会从移位寄存器转移到 USART_RDR 寄存器。
- 单字节通信的时候不会产生中断。然而与这一位同时上升的 RXNE 是会产生中断的。在多缓冲区通信的情况下，只要 USART_CR3 中的 EIE 位是高就会产生中断。

将 ICR 寄存器中的 NFCF 位置 1 就可以清除 NF 标志。

注：在智能卡、IrDA 和 LIN 模式下，不能用 8 倍过采样的方式。在那些模式下，OVER8 位会由硬件强制为 0。

7．帧错误

由于没有同步上或大量噪音的原因，停止位没有在预期的时间上接收和识别出来，会检测到帧错误。当帧错误被检测到时，具有如下特征。

- FE 位被硬件置 1。
- 有问题的数据仍会从移位寄存器转移到 USART_RDR 寄存器。
- 单字节通信的时候不会产生中断。然而与这一位同时上升的 RXNE 是会产生中断的。在多缓冲区通信的情况下，只要 USART_CR3 中的 EIE 位是高就会产生中断。

将 USART_ICR 寄存器中的 FECF 置 1 就可以清除 FE 标志。

8．接收期间的可配置的停止位

被接收的停止位的个数可以通过 USART_CR2 的控制位来配置。在正常模式时，可以是 1 个或 2 个，在智能卡模式里可能是 1.5 个。

- 1 个停止位：对 1 个停止位的采样在第 8、第 9 和第 10 个采样点上进行。
- 1.5 个停止位（智能卡模式）：当以智能卡模式发送时，器件必须检查数据是否被正确地发送出去。所以接收器功能块必须被激活（USART_CR1 寄存器中的 RE =1），并且在停止位的发送期间采样数据线上的信号。如果出现校验错误，智能卡会在发送方采样 NACK 信号时，即总线上停止位对应的时间内时，拉低数据线，以此表示出现了帧错误。FE 在 1.5 个停止位结束时和 RXNE 一起被置 1。对 1.5 个停止位的采样是在第 16、第 17 和第 18 个采样点进行的。1.5 个的停止位可以被分成两部分：一个是 0.5 个时钟周期，期间不做任何事情；随后是 1 个时钟周期的停止位，在这段时间的中点处采样。
- 2 个停止位：对 2 个停止位的采样是在第一停止位的第 8、第 9 和第 10 个采样点完成的。如果第一个停止位期间检测到一个帧错误，帧错误标志将被设置。第二个停止位不再检查帧错误。在第一个停止位结束时 RXNE 标志将被设置。

7.3.5 多机通信

可以将多个 USART 连接成一个网络来实现多机通信。例如，某个 USART 设备可以是主，它的 TX 输出和其他 USART 从设备的 RX 输入相连接；USART 从设备各自的 TX 输出逻辑地与在一起，并且和主设备的 RX 输入相连接，如图 7-8 所示。

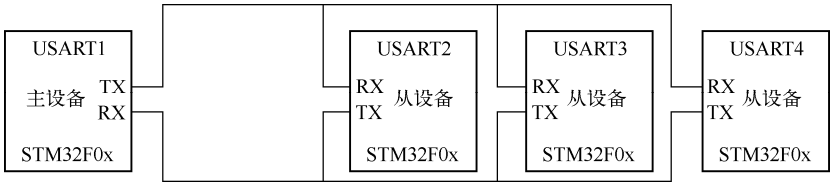


图 7-8　多机通信

在多处理器配置中，我们通常希望只有被寻址的接收者才被激活，来接收随后的数据，这样就可以减少由未被寻址的接收器的参与带来的多余的 USART 服务开销。未被寻址的设备可启用其静默功能进入静默模式。要使用静默模式功能，USART_CR1 寄存器的 MME 位必须被置 1。在静默模式里，具有如下特征。

- 任何接收状态位都不会被置 1。
- 所有接收中断被禁止。
- USART_CR1 寄存器中的 RWU 位被置 1。RWU 可以被硬件自动控制或在某个条件下由软件通过 USART_RQR 寄存器的 MMRQ 位写入。

根据 USART_CR1 寄存器中的 WAKE 位状态，USART 可以用两种方法进入或退出静默模式。

- 如果 WAKE 位为 0，进行空闲总线检测。
- 如果 WAKE 位为 1，进行地址标记检测。

1．空闲总线检测（WAKE=0）

当 MMRQ 位被置 1，并且 RWU 被自动置 1 时，USART 进入静默模式。检测到一个空闲帧时 USART 被唤醒。RWU 位被硬件清零，但 USART_ISR 寄存器中的 IDEL 位没有被置 1。

注：（1）如果在空闲字符已经过去之后才置 1MMRQ，将不会退出静默模式（RWU 没被置 1）。

（2）如果在线路空闲期间激活 USART，在一个空闲帧时长之后才会检测到空闲状态（不仅仅是在收到一个数据帧之后）。

2．4 位/7 位地址标记检测（WAKE=1）

在这个模式里，如果 MSB 是 1，该字节被认为是地址，否则被认为是数据。在一个地址字节中，目标接收器的地址被放在 4 位或 7 位的位域中。用 ADDM7 位来选择是用 4 位地址还是用 7 位地址。这个 4 位或 7 位地址被接收器同它自己地址做比较，接收器的地址被设置在 USART_CR2 寄存器的 ADD 域。

注：在 7 位和 9 位数据模式下，地址检测分别按 6 位和 8 位地址（ADD[5:0]和 ADD[7:0]）操作。

如果接收到的字节与它的编程地址不匹配时，USART 进入静默模式。此时，硬件将 RWU 位置 1。当 USART 进到静默模式后，接收字节时既不会影响 RXNE 标志，也不会产生中断或发出 DMA 请求。将 MMRQ 置 1 也会令 USART 进入静默模式。这时 RWU 位也被自动置 1。

如果接收到的字节与它的编程地址匹配，USART 将退出静默模式。然后 RWU 位被清零，后续的字节会被正常接收。从 RWU 位被清零开始，RXNE 位会因为地址字节的接收而被置 1。

7.3.6　Modbus 通信

Modbus 是由施耐德创建的一个串行通信协议，通过数据流的判断，以及格式规定实现串行通信。Modbus 分为 RTU 和 ASCII 形式。USART 提供对 Modbus/RTU 和 Modbus/ASCII 协议实现的基本支持。USART 提供对块尾检测的基本支持，无须软件

的经常性介入。

1．Modbus/RTU

这个模式下，块尾一般为一个超过 2 个字符长度的静默阶段，通过一个可设置的超时长度功能来实现。超时功能和相应的中断必须通过 USART_CR2 寄存器中的 RTOEN 位和 USART_CR1 寄存器中的 RTOIE 位来打开。RTO 寄存器中要填入一个与超时长度相当的数字（例如 2 个字符长度为 22 位长）。当接收线路保持空闲阶段达到这个长度时，在最后一个停止位被收到之后，会产生一个中断，表示当前的块接收已经完毕。

2．Modbus/ASCII

在这个模式，块尾一般为回车字符（CR/LF）串。USART 用字符匹配功能实现这个机制。将 LF 的 ASCII 码写到 ADD[7:0]区域，然后打开字符匹配中断（CMIE=1），那么软件就会在收到 LF 字符后或者能够在 DMA 缓冲区中找到 CR/LF 字符时得到提示。

7.3.7 LIN（本地互联网络）模式

本节只针对支持 LIN 模式的芯片。

LIN 是用于汽车中的串行网络通信协议。相比 CAN 协议，成本连接低，所以经常被作为汽车中的辅助协议，采用的是一主多从的广播形式。典型的 LIN 总线应用是汽车中的联合装配单元，如门、方向盘、座椅、空调、照明灯、湿度传感器，交流发电机等。对于这些成本比较敏感的单元，LIN 可以使机械元件如智能传感器、制动器或光敏器件得到较广泛的使用，很容易地连接到汽车网络中并可方便地维护。

TJA1020 是 LIN 主/从协议控制器和 LIN 本地互联网络物理总线之间的接口，主要用作车辆中的副网络。使用的波特率为 2.4～20kbit/s。支持 LIN 模式的 STM32F0x 可与 TJA1020 构成 LIN 网络。

STM32F0x 的 LIN 模式是通过设置 USART_CR2 寄存器的 LINEN 位选择的。在 LIN 模式下，下列位必须保持为 0。

- USART_CR2 寄存器的 CLKEN 位。
- USART_CR3 寄存器的 STOP[1:0]、SCEN、HDSEL 和 IREN。

1．LIN 发送

LIN 的主机发送和常规的 USART 发送相同，但包含下列区别。

- 清零 M 位以配置 8 位字长。
- LINEN 位置 1 以进入 LIN 模式。这时，置 1，SBK 将发送 13 位“0”作为断开符号；然后发一位“1”，以允许对下一个开始位的检测。

2．LIN 接收

当 LIN 模式被使能时，断开符号检测电路被激活。该检测完全独立于 USART 接收器。不管是在总线空闲时还是在发送某数据帧期间，断开符号只要一出现就能检测到。

一旦接收器被激活（USART_CR1 的 RE=1），电路就开始监测 RX 上的起始信号。监测起始位的方法同检测断开符号或数据是一样的。当起始位被检测到后，电路对每个接下来的位，在每个位的第 8、9、10 个过采样时钟点上进行采样，就像针对数据一样。如果 10 个（当 USART_CR2 的 LBDL=0）或 11 个（当 USART_CR2 的 LBDL=1）连续位都是“0”，并且又跟着一个定界符，USART_SR 的 LBD 标志就会被置 1。如果 LBDIE 位为 1，还会产生中

断。在确认断开符号前，要检查定界符，因为它表示 RX 线已经回到高电平。

如果在第 10 或 11 个采样点之前采样到了“1”，检测电路取消当前检测并重新寻找起始位。如果 LIN 模式被禁止，接收器继续工作像正常 USART 一样，但不再考虑检测断开符号。

如果 LIN 模式被激活（LINEN=1），只要一发生帧错误（例如：停止位检测到“0”，这种情况出现在断开符号被接收到的时候），接收器就停止，直到断开符号检测电路接收到一个“1”（这种情况发生于断开符号没有完整地发出来）或一个定界符（这种情况发生于已经检测到一个完整的断开符号）。

7.3.8　USART 同步模式

同步模式与异步模式的区别：主设备对外提供时钟，从设备不是依靠提前双方约定的波特率去检测信号，而是依靠主设备发送过来的时钟信号。SPI 与 I^2C 均属于此类。图 7-9 是 USART 的同步模式用法。

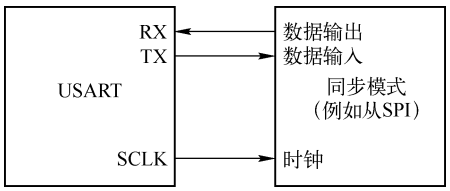


图 7-9　USART 同步模式使用

通过在 USART_CR2 寄存器的 CLKEN 位上写 1 来选择同步模式。在同步模式下，下列位必须保持为 0。

- USART_CR2 寄存器中的 LINEN 位；
- USART_CR3 寄存器中的 SCEN、HDSEL 和 IREN 位。

USART 允许用户以主模式方式控制双向同步串行通信。SCLK 脚是 USART 发送器时钟的输出。在起始位和停止位期间，SCLK 脚上没有时钟脉冲。USART_CR2 寄存器中 LBCL 位的状态决定了在最后一个有效数据位期间产生或不产生时钟脉冲。USART_CR2 寄存器的 CPOL 位允许用户选择时钟极性，USART_CR2 寄存器的 CPHA 位允许用户选择外部时钟的相位。

在总线空闲期间，实际数据到来之前以及发送断开符号的时候，外部 SCLK 时钟不被激活。同步模式时，USART 发送器和在异步模式里工作一模一样。但是因为 SCLK 与 TX 同步（根据 CPOL 和 CPHA），所以 TX 上的数据是随 SCLK 同步发出的。

同步模式的 USART 接收器工作方式与异步模式不同。如果 RE=1，数据在 SCLK 上采样（根据 CPOL 和 CPHA 决定在上升沿还是下降沿），不需要任何的过采样，但必须考虑建立时间和持续时间（取决于波特率，1/16 位时间）。

注：（1）SCLK 脚同 TX 脚是协同工作的。因而，只有在使能了发送器（TE = 1），并且发送数据时（写入数据至 USART_DR 寄存器）才提供时钟。这意味着在没有发送数据时是不可能接收一个同步数据的。

（2）应该在发送器和接收器都被禁止时（UE=0）去改变 LBCL、CPOL 和 CPHA 位的配置，这能保证时钟脉冲功能的正确性。

7.3.9 单线半双工通信

单线半双工模式通过设置 USART_CR3 寄存器的 HDSEL 位来选择。在本模式下，下列位必须保持为 0。

- USART_CR2 寄存器的 LINEN 和 CLKEN 位；
- USART_CR3 寄存器的 SCEN 和 IREN 位。

USART 可以配置成遵循单线半双工协议。在单线半双工模式下，TX 和 RX 引脚在芯片在内部是连在一起的。使用控制位“HALF DUPLEX SEL”（USART_CR3 中的 HDSEL 位）选择半双工和全双工通信。

当 HDSEL 为“1”时，TX 和 RX 状态如下。

- TX 和 RX 引脚在芯片在内部是连在一起的；
- RX 不再被使用；
- 当没有数据传输时，TX 总是被释放。因此，它在空闲状态的或接收状态时表现为一个标准 I/O 口。这就意味该 I/O 在不被 USART 驱动时，必须配置成悬空输入（或开漏的输出高）。

除此以外，通信与正常 USART 模式类似。由软件来管理线上的冲突（例如通过使用一个中央仲裁器），特别的是，发送从不会被硬件所阻碍。当 TE 位被设置时，只要数据一写到数据寄存器上，发送就会开始。

7.3.10 RS-232 硬件流控制和 RS-485 驱动使能

利用 nCTS 输入和 nRTS 输出可以控制两个设备间的串行数据流。图 7-10 显示了这种模式下连接两个设备。通过将 UASRT_CR3 中的 RTSE 和 CTSE 置 1，可以分别独立地使能 RS-232 的 RTS 和 CTS 流控制。

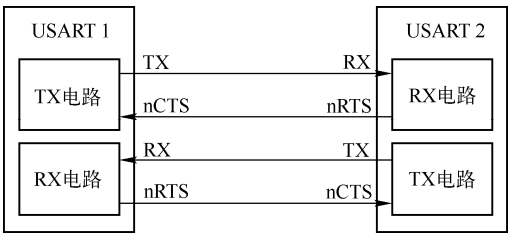


图 7-10　两个 USART 之间的硬件流控制

1．RS-232 的 RTS 流控制

如果 RTS 流控制被使能（RTSE=1），只要 USART 接收器准备好接收新的数据，nRTS 就变成有效（接低电平）。当接收寄存器内的数据未被取走时，nRTS 被释放，由此表明希望在当前帧结束时停止数据传输。图 7-11 是一个启用 RTS 流控制的通信的例子。

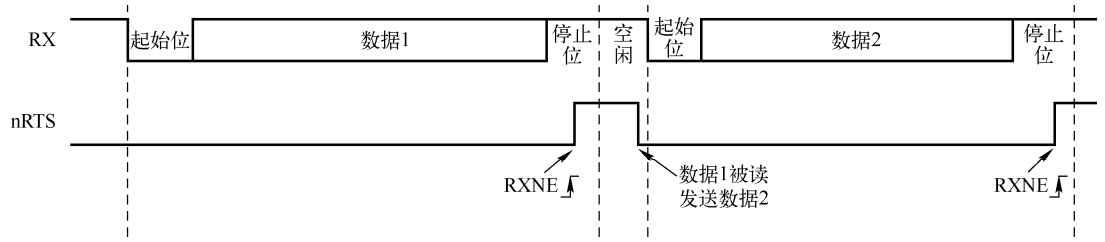


图 7-11　RTS 流控

2．CTS 流控制

如果 CTS 流控制被使能（CTSE=1），发送器在发送下一帧前检查 nCTS 输入。如果 nCTS 有效（被拉成低电平），则下一个数据被发送（假设那个数据是准备发送的，也就是如果 TXE=0），否则下一帧数据不被发出去。若 nCTS 在传输期间被变成无效，当前的传输完成后才停止发送。

当 CTSE=1 时，只要 nCTS 输入一变换状态，硬件就自动设置 CTSIF 状态位。它表明接收器是否准备好进行通信。如果设置了 USART_CT3 寄存器的 CTSIE 位，则产生中断。

图 7-12 是一个启用 CTS 流控制的通信的例子。

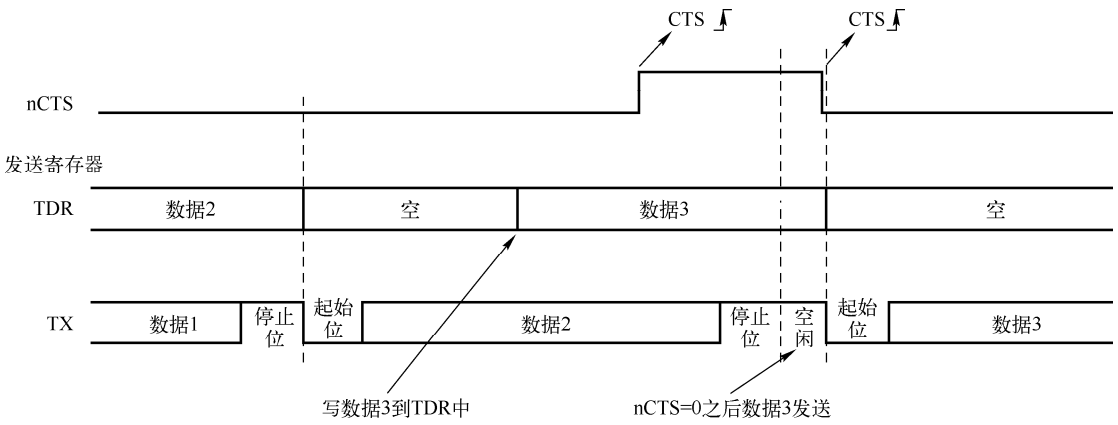


图 7-12　CTS 流控制

注：nCTS 必须比当前字符的结束提前至少 3 个 USART 时钟周期做动作。另外需要注意的是，对于比 2 个 PCLK 周期短的脉冲，CTSCF 标志不一定能够置 1。

3．RS-485 驱动使能

驱动使能功能用 USART_CR3 控制寄存器的 DEM 位置 1 来打开。它允许用户通过 DE（驱动使能）信号来激活外部收发器的控制端。提前时间的意思是驱动使能信号和第一个字节的起始位之间的时间间隔。这个时间可以在 USART_CR1 控制寄存器的 DEAT[4:0]域中设置。滞后时间是一个发送消息的最后一个字节的停止位和释放 DE 信号之间的时间间隔。这个时间可以在 USART_CR1 控制寄存器的 DEDT[4:0]域中设置。DE 信号的极性则可以通过 USART_CR3 控制寄存器中的 DEP 位进行选择。

7.4 USART 中断

表 7-2 是 USART 的中断说明。

表 7-2 USART 中断请求

中 断 事 件	事 件 标 志	使能控制位
发送数据寄存器空	TXE	TXEIE
CTS 中断	CTSIF	CTSIE
发送完成	TC	TCIE
接收数据寄存器非空（有数据可读）	RXNE	RXNEIE
溢出错误检测	ORE	
空闲线检测	空闲	IDLEIE
奇偶错误	PE	PEIE
LIN 断开	LBDF	LBDIE
噪声标志，溢出错误和多缓冲区通信中的帧错误	NF or ORE or FE	EIE
字符匹配	CMF	CMIE
接收超时错误	RTOF	RTOIE
发现块尾	EOBF	EOBIE
从 Stop 模式唤醒	WUF(1)	WUFIE

USART 中断信号全部连接到同一个中断向量（见图 7-13）。

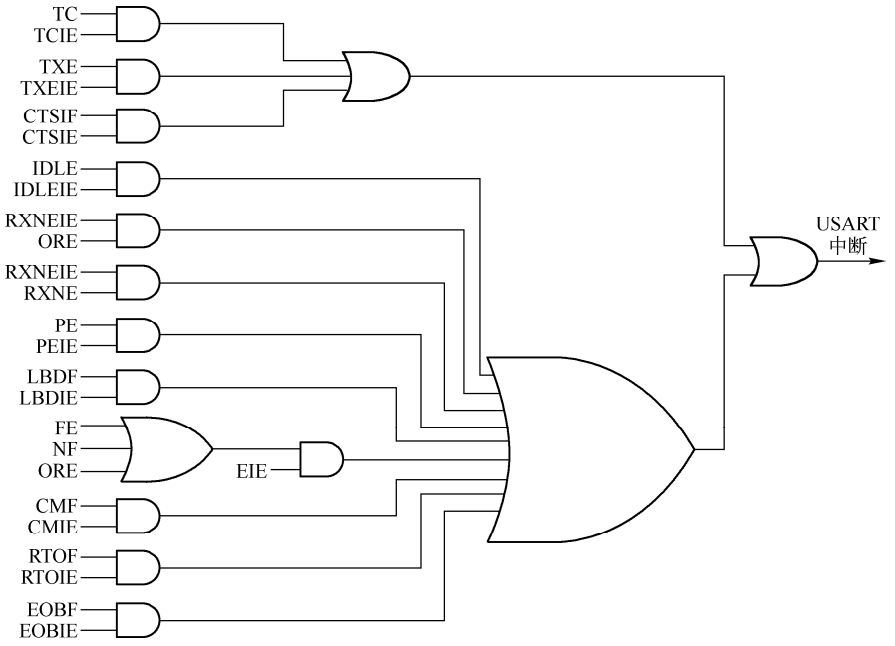


图 7-13 USART 中断镜像图

- 发送期间：发送完成，CTS，发送数据寄存器空或帧错误（智能卡模式下）中断。
- 接收期间：空闲线检测，溢出错误，接收数据寄存器非空，校验错误，LIN 断开检测，噪声标志（仅在多缓冲区通信时），帧错误（仅在多缓冲区通信），字符匹配，等等。
- 如果设置了相应的使能控制位，这些事件都可以引起中断。

7.5　USART 固件库函数

USART 的固件库根据 USART 分为初始化配置、停止模式函数、自动波特率函数、数据收发函数、多处理器通信、LIN 模式函数、半双工模式函数、智能卡模式、IrDA 模式函数、RS-485 模式。表 7-3 是函数的说明。

表 7-3　USART 固件库函数

组	函　数	功 能 描 述
初始化和配置函数	void USART_DeInit(USART_TypeDef* USARTx)	恢复USARTx外设寄存器为默认初始状态
	void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct)	由USART_InitStruct参数初始化USARTx外设
	void USART_StructInit(USART_InitTypeDef* USART_InitStruct)	默认值填充USART_InitStruct
	void USART_ClockInit(USART_TypeDef* USARTx, USART_ClockInitTypeDef* USART_ClockInitStruct)	使用输入参数初始化USARTx外设时钟
	void USART_ClockStructInit(USART_ClockInitTypeDef* USART_ClockInitStruct)	默认值设置USART_ClockInitStruct
	void USART_Cmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用USART
	void USART_DirectionModeCmd(USART_TypeDef* USARTx, uint32_t USART_DirectionMode, FunctionalState NewState)	使能或禁用USART的发送器和接收器
	void USART_SetPrescaler(USART_TypeDef* USARTx, uint8_t USART_Prescaler)（STM32F030 不可用）	设置系统分频
	void USART_OverSampling8Cmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用USART的8X过采样
	void USART_OneBitMethodCmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用USART的单位采样
	void USART_MSBFirstCmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用单位在前
	void USART_DataInvCmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用USART的二进制数据反向
	void USART_InvPinCmd(USART_TypeDef* USARTx, uint32_t USART_InvPin, FunctionalState NewState)	使能或禁用引脚的有效电平反向
	void USART_SWAPPinCmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用Tx/Rx引脚互换
	void USART_ReceiverTimeOutCmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用接收器超时机制
	void USART_SetReceiverTimeOut(USART_TypeDef* USARTx, uint32_t USART_ReceiverTimeOut)	设置接收器超时间隔

续表

组	函　数	功 能 描 述
停止模式函数	void USART_STOPModeCmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用停止模式
	void USART_StopModeWakeUpSourceConfig(USART_TypeDef* USARTx, uint32_t USART_WakeUpSource)（STM32F030不可用）	配置停止模式唤醒源
自动波特率函数	void USART_AutoBaudRateCmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用自动波特率
	void USART_AutoBaudRateConfig(USART_TypeDef* USARTx, uint32_t USART_AutoBaudRate)	选择自动波特率检测方法
数据收发函数	void USART_SendData(USART_TypeDef* USARTx, uint16_t Data)	发送数据
	uint16_t USART_ReceiveData(USART_TypeDef* USARTx)	接收数据
多处理器通信	void USART_SetAddress(USART_TypeDef* USARTx, uint8_t USART_Address)	设置USART节点的地址
	void USART_MuteModeWakeUpConfig(USART_TypeDef* USARTx, uint32_t USART_WakeUp)	设置静默模式下的唤醒方法
	void USART_MuteModeCmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能静默模式
	void USART_AddressDetectionConfig(USART_TypeDef* USARTx, uint32_t USART_AddressLength)	地址检测配置
LIN模式函数	void USART_LINBreakDetectLengthConfig(USART_TypeDef* USARTx, uint32_t USART_LINBreakDetectLength)（STM32F030不可用）	设置LIN断开检测长度
	void USART_LINCmd(USART_TypeDef* USARTx, FunctionalState NewState)（STM32F030不可用）	使能或禁用LIN网
半双工模式函数	void USART_HalfDuplexCmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用半双工模式
智能卡模式	void USART_SmartCardCmd(USART_TypeDef* USARTx, FunctionalState NewState)（STM32F030不可用）	使能或禁用USART的智能卡模式
	void USART_SmartCardNACKCmd(USART_TypeDef* USARTx, FunctionalState NewState)（STM32F030不可用）	使能或禁用智能卡的NACK发送
	void USART_SetGuardTime(USART_TypeDef* USARTx, uint8_t USART_GuardTime)（STM32F030不可用）	设置保护时间
	void USART_SetAutoRetryCount(USART_TypeDef* USARTx, uint8_t USART_AutoCount)（STM32F030不可用）	设置智能卡收发自动重试次数
	void USART_SetBlockLength(USART_TypeDef* USARTx, uint8_t USART_BlockLength)（STM32F030不可用）	设置智能卡的块长度
IrDA模式函数	void USART_IrDAConfig(USART_TypeDef* USARTx, uint32_t USART_IrDAMode)（STM32F030不可用）	配置USART的IrDA接口
	void USART_IrDACmd(USART_TypeDef* USARTx, FunctionalState NewState)（STM32F030不可用）	使能或禁用
RS-485模式	void USART_DECmd(USART_TypeDef* USARTx, FunctionalState NewState)	使能或禁用USART的DE功能
	void USART_DEPolarityConfig(USART_TypeDef* USARTx, uint32_t USART_DEPolarity)	配置USART的DE极性
	void USART_SetDEAssertionTime(USART_TypeDef* USARTx, uint32_t USART_DEAssertionTime)	设置DE的提前确认时间
	void USART_SetDEDeassertionTime(USART_TypeDef* USARTx, uint32_t USART_DEDeassertionTime)	设置DE的延迟确认时间
DMA传输管理函数	void USART_DMACmd(USART_TypeDef* USARTx, uint32_t USART_DMAReq, FunctionalState NewState)	使能或禁用DMA接口
	void USART_DMAReceptionErrorConfig(USART_TypeDef* USARTx, uint32_t USART_DMAOnError)	在接收错误时使能或禁用DMA接口

续表

组	函　　数	功 能 描 述
中断和标志管理	void USART_ITConfig(USART_TypeDef* USARTx, uint32_t USART_IT, FunctionalState NewState)	USART的中断配置
	void USART_RequestCmd(USART_TypeDef* USARTx, uint32_t USART_Request, FunctionalState NewState)	使能USART的特定请求
	void USART_OverrunDetectionConfig(USART_TypeDef* USARTx, uint32_t USART_OVRDetection)	溢出检测配置
	FlagStatus USART_GetFlagStatus(USART_TypeDef* USARTx, uint32_t USART_FLAG)	读标志状态
	void USART_ClearFlag(USART_TypeDef* USARTx, uint32_t USART_FLAG)	清标志
	ITStatus USART_GetITStatus(USART_TypeDef* USARTx, uint32_t USART_IT)	读中断状态
	void USART_ClearITPendingBit(USART_TypeDef* USARTx, uint32_t USART_IT)	清中断挂起位

USART 固件库的用法如下。

- RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE)：使能外设时钟或者通过。
- RCC_APB1PeriphClockCmd(RCC_APB1Periph_USARTx, ENABLE)：使能 USART2 和 USART3。
- 根据 USART 模式，使用 RCC_AHBPeriphClockCmd()函数使能 GPIO 时钟（IO 端口可能是 TX、RX、CTS、SCLK）。
- 外设功能如下。
  - 通过 GPIO_PinAFConfig()函数连接端口到希望的外设功能。
  - 通过 GPIO_InitStruct->GPIO_Mode = GPIO_Mode_AF 配置相应的复用功能。
  - 通过 GPIO_PuPd, GPIO_OType 和 GPIO_Speed 成员选择类型、上拉/下拉和输出速率。
  - 调用 GPIO_InitStruct 函数。
- 通过 USART_Init 设置波特率、字长、停止位、奇偶校验、硬件流控制和模式（发送/接收）。
- 针对同步模式，使用 USART_ClockInit()函数使能时钟和编写奇偶相以及停止位。
- 如采用中断模式，使用 USART_ITConfig()与 NVIC_Init()使能 NVIC 以及相应的中断。
- 当使用 DMA 模式时，使用 DMA_Init()函数配置 DMA，使用 USART_DMACmd()函数激活所需的通道请求，使用 DMA_Cmd()函数使能 DMA。

另外在 STM32F0 的固件库中提供了一种超时检测机制（见 STM32F0xx_StdPeriph_Examples\USART\USART_TwoBoards\DataExchangeInterrupt）。该方式是通过 SysTick 定时器递减计数器的方法检测发送或接收超时，从而判断发送或者接收是否结束。该方式类似 Modbus 中的超时检测机制。

printf 函数是 C 语言的一个很常用函数，但在嵌入式系统，该函数需要自己实现。在 mdk 中用 printf，需要同时重定义 fputc 函数和避免使用 semihosting（半主机模式），标准库函数的默认输出设备是显示器，要实现在串口或 LCD 输出，必须重定义标准库函数里调用的与输出设备相关的函数。例如：printf 输出到串口，需要将 fputc 里面的输出指向串口（重定向），方法如下。

```
#ifdef __GNUC__
  /* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
     set to 'Yes') calls __io_putchar() */
  #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
  #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */

PUTCHAR_PROTOTYPE
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the USART */
  USART_SendData(EVAL_COM1, (uint8_t) ch);

  /* Loop until the end of transmission */
  while (USART_GetFlagStatus(EVAL_COM1, USART_FLAG_TC) == RESET)
  {}

  return ch;
}
```

因 printf()使用了半主机模式，使用标准库会导致程序无法运行，可使用 MicroLib。即在 MDK 中的 Options for Target 界面中选择 Target 后，在 Code Generation 一栏选择 Use MicroLIB。

由于汉语是我们的母语，通过串口显示信息，会很习惯地使用汉字。在代码中编译汉字对程序本身无任何影响，MDK 将汉字编译成两个字节。问题的关键是显示部分，即显示终端是否支持。假如使用的串口显示屏，例如国内迪文科技、周立功的串口屏，如何显示串口发送过来的汉字内容，取决串口屏的字库配置信息。如果串口连接的是 PC 超级终端，需要将 PC 的超级终端设置成能够显示汉字方式。由于 Windows 7 不再提供超级终端，推荐使用 Putty 的超级终端。该终端显示汉字的配置如图 7-14 与图 7-15 所示。

图 7-14　Putty 字体设置

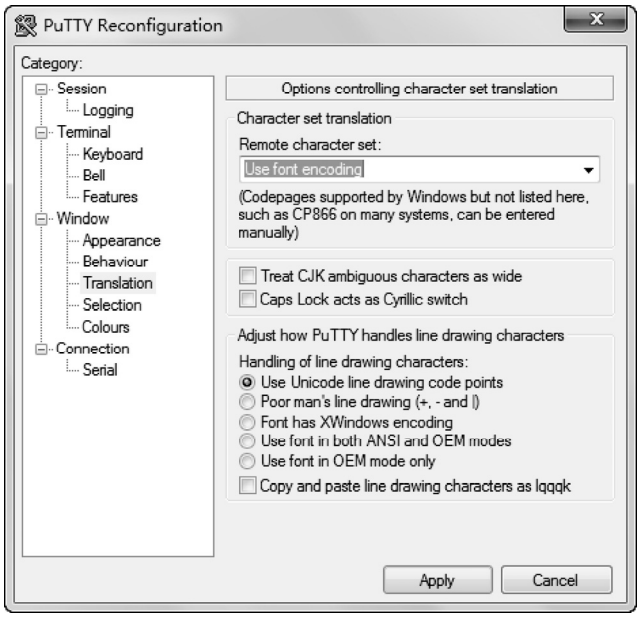


图 7-15　Putty 编码设置

7.6　基于 USART 实现的多个通信标准

USART 是一种同步异步通信模式，基于 USART 的硬件标准有 RS-232、RS-485、RS-422、LIN 网、IO-Link 网。另外有基于串行通信机制的软件层次的协议标准：Modbus 通信协议、Profibus。另有一个很特殊之处，由于现在多数 PC 取消了 RS-232 接口，目前较为流行的方式是通过 USB 接口转换为 RS-232 接口或者 TTL 电平方式。常见的 USB 转串口芯片 FT232RL/BL、PL2303、CP2102 等将 USB 信号转换成 TTL 电平，直接与 STM32F0 的相应引脚相连，在 PC 端安装相应的驱动软件之后，可在 PC 的设备管理器中看到相应的 COM。

RS-232 标准的优点是传输线少，配线简单，发送距离可以较远（对比 TTL 电平）。由于设备供电电源的不同，±5V、±10V、±12V 和±15V 这样的电平都是可能的。图 7-16 是典型 RS-232 实现。SP3232E 是电平转换，用于将 TTL 电平转换成 RS-232 电平。这类芯片很多，选型时间根据成本以及需几路 RS-232 选型。如果用于工业场合以及医疗场合，建议选用带有 ESD 防护功能的 RS-232 电平转换芯片。STM32F0x 的 USART 引脚连接到图 7-16 中的 USART_TxD 与 USAT_RXD 引脚即可。

RS-485 是 2 线、半双工、多点通信的标准，但电气特性与 RS-232 不一样。缆线两端的电压差值用来表示传递信号，1 端的电压标识为逻辑 1，另一端标识为逻辑 0。两端的电压差最小为 0.2V 以上时有效，任何不大于 12V 或者不小于−7V 的差值对接收端都被认为是正确的。图 7-17 是 RS-485 的电路实现。其中电平转换可选用 MAX485CSA 或者 MAX13487EESA。半双工形式的 RS-485 收发过程需要一个使能端进行收发控制。选用 MAX485CSA 则通过 P1 端子将 U1 的使能端端接到 PA1，进而连接到 STM32F0x，由软件控

制收发转换。如果使用 MAX13487EESA 芯片，P1 连接到 VDD3.3 上即可，实现自动收发，但成本稍高。图 7-17 中上拉电阻 R1 与下拉电阻 R4 是为了保证在所有 485 均接收的时候总线上保持一个确定的电平，否则所有 485 均在高阻输入态的时候，总线电压状态不定。R2 与 R5 为限流电阻。R3 用于 RS-485 连接时末端的匹配电阻，短距离传输不需要。LD1 与 LD2 起到 ESD 与浪涌保护功能。如果能够承受成本上升，可以在 UART_RX 与 UART_TX 连接 STM32F0x 引脚之前，分别连接一个 6N137 快速光耦，起到隔离作用。

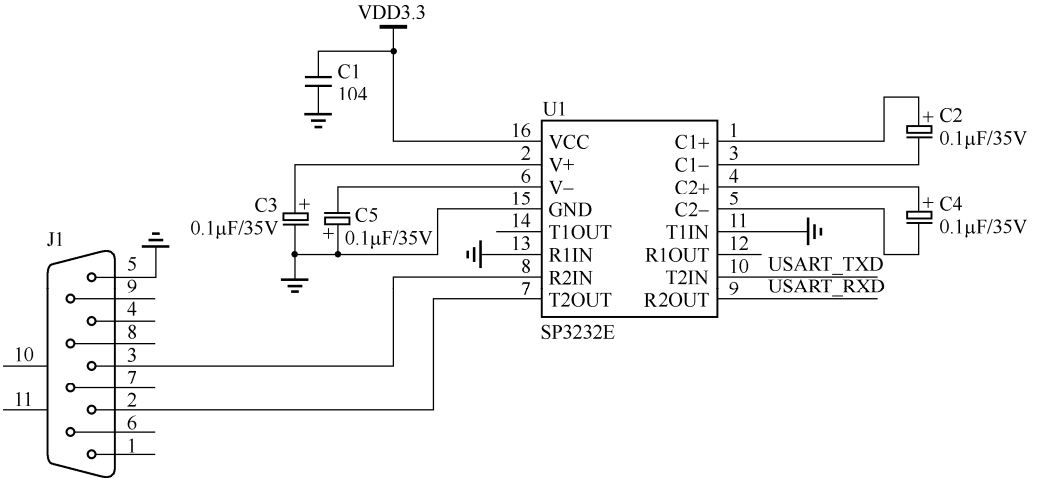


图 7-16　RS-232 电路实现

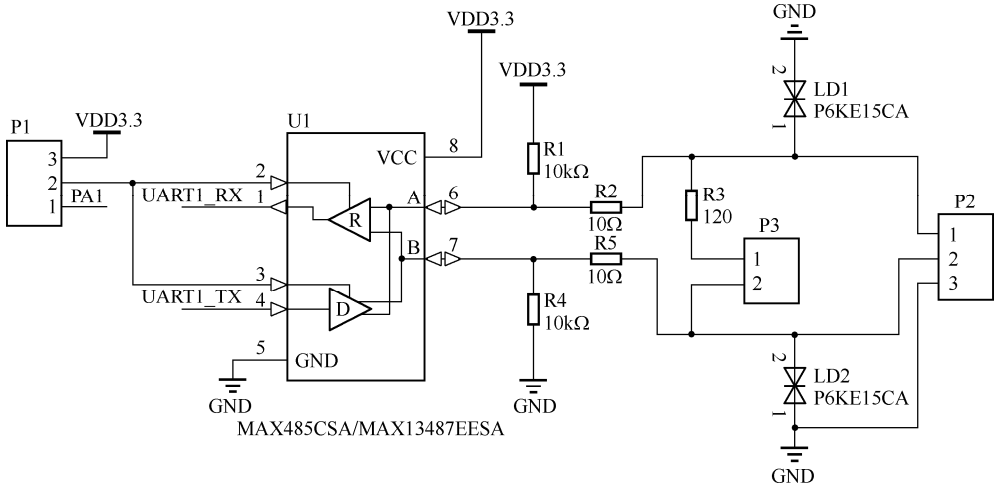


图 7-17　RS-485 实现电路

IO-Link 被称为直达传感器的通信，是一种点对点连接方式，它通过 3 线制，使用非屏蔽标准电缆连接至传统型和智能型传感器/执行器。IO-Link 可向后兼容所有的 DI/DO 传感器/执行器，采用 3 线的 24VDC 连接方式。图 7-18 是 IO-Link 实现原理图，其中 ST 的 L6360 是 IO-Link 协议主发送芯片。

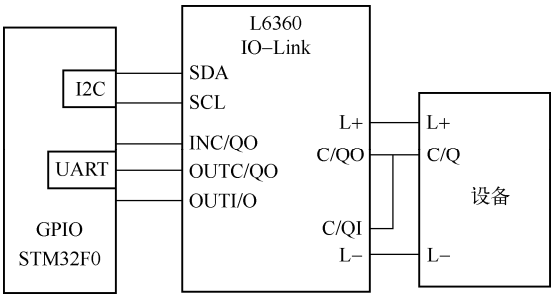


图 7-18　IO-Link 实现原理图

基于汽车中的节点数持续增加，为解决系统复杂性与低成本的矛盾，创立了 LIN（局部互联网络）总线。该标准涵盖了传输协议介质定义工具和应用编程接口，保证了网络节点硬件/软件的互用性。LIN 总线是单线（12V 信号总线）总线；低成本，基于通用 UART 接口；不需要改变 LIN 从节点的硬件和软件就可以在网络上增加节点；单主控器/多从设备模式，无须仲裁机制；传输速率最高可达 20kbit/s。ST 公司的 L9638 收发器以及 NXP 公司的 TJA1020 是 LIN 总线的收发器。图 7-19 是 LIN 网络原理图。

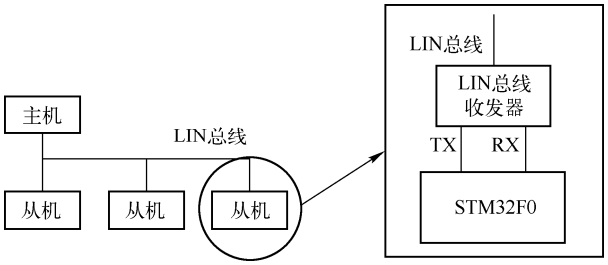


图 7-19　LIN 网原理图

7.7　接收不定长数据实例

在本章提到了 STM32F0 有多个 STM32F1 不具备的功能，其中对 Modbus 协议的超时检测功能是一个非常优秀的功能。下面代码演示了如何使用该功能。

```
//发送数据，Length 为发送数据长度
void UART_Send(uint8_t *Buffer, uint32_t Length)
{
while(Length != 0)
{
    while(!((USART1->ISR)&(1<<7)));//等待发送完
    USART1->TDR= *Buffer;
    Buffer++;
    Length--;
}
}
```

```
void USART_Configuration(void)//串口初始化函数
  {
        USART_InitTypeDef USART_InitStructure;
      GPIO_InitTypeDef    GPIO_InitStructure;
      NVIC_InitTypeDef NVIC_InitStructure;

      RCC_AHBPeriphClockCmd( RCC_AHBPeriph_GPIOA, ENABLE);
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE );

      GPIO_PinAFConfig(GPIOA,GPIO_PinSource2,GPIO_AF_1);
      GPIO_PinAFConfig(GPIOA,GPIO_PinSource3,GPIO_AF_1);

      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2|GPIO_Pin_3;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
      GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
      GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_Init(GPIOA, &GPIO_InitStructure);

      USART_InitStructure.USART_BaudRate = 9600;//设置串口波特率
      USART_InitStructure.USART_WordLength = USART_WordLength_8b;//设置数据位
      USART_InitStructure.USART_StopBits = USART_StopBits_1;//设置停止位
      USART_InitStructure.USART_Parity = USART_Parity_No;//设置效验位
      USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
                                          //设置流控制
      USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
                                          //设置工作模式
      USART_Init(USART1, &USART_InitStructure);          //配置入结构体
          USART_SWAPPinCmd(USART1,ENABLE);
                                        //根据是否将 TX 与 RX 交互决定是否启动该行代码
      USART_Cmd(USART1, ENABLE);//使能串口 1
            NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
            NVIC_InitStructure.NVIC_IRQChannelPriority = 0;
            NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
            NVIC_Init(&NVIC_InitStructure);
}
```

接收中断与超时中断在 stm32f0xx_it.c 文件中。

```
/**接收中断 */
void USART1_IRQHandler(void)
{extern   unsigned int end;
//接收到一个完整数据包；
     if (USART_GetITStatus(USART1, USART_IT_RTO) == SET)
 {
 end =1;
 USART_ClearITPendingBit(USART1, USART_IT_RTO);//该中断标志需软件清除
```

```
    }
      /* 接收中断 */
      if (USART_GetITStatus(USART1, USART_IT_RXNE) == SET)
      {
            RxBuffer[RxIndex++] = USART_ReceiveData(USARTx);//读取数据时，清除接收中断；
      }
}
```

USART_Config 函数是完整的 USART 配置函数，与 7.5 节提到的固件库用法相符。

在 main 函数中调用 USART_Config 函数之后，设置接收中断与接收超时中断。接收超时中断需要先通过 USART_ReceiverTimeOutCmd (USART1, ENABLE)开启接收超时功能，而后通过 USART_SetReceiverTimeOut(USART1,20)接收超时时间。本例中采用 2 个字符超时（一个字符数据为 10 位，即 1 个起始位、1 个停止位、无奇偶校）。

数据发送未采用中断模式发送数据，采用循环模式发送数据。在每次发送数据之前读取 USART 的发送寄存器是否为空。

USART1_IRQHandler 是 USART 中断，用来处理接收数据以及接收超时中断。USARTx_ISR 寄存器中的 RTOF 与 RXNE 在使用上稍微有些差别。读取 USARTx_RDR 寄存器会清除 RXNE 位（当然也可软件清除该位），但 RTOF 位置需通过软件清除方式（对应代码是 USART_ClearITPendingBit(USARTx, USART_IT_RTO)）。

该程序可通过串口调试小助手进行观察。通过发送不同长度的数据，会观察到收到一包完整数据。如果是将代码中的接收中断屏蔽，会观察到接收数据存在断续现象。

针对本节的程序有两个思考题。

（1）通过循环方式发送数据有什么不良影响？

提示：本节采用接收超时中断模式接收数据。发送方发送数据一次是发送整包数据。如果存在中断程序执行时间超过本节设置的时间间隔，在主循环中发送数据会存在数据间隔。

（2）将串口调试小助手设置成自动发送。将发送时间间隔不断调小，观察一下收到的数据有什么特点。

提示：本节设置了数据包的时间间隔为 2 个字符长度，发送数据采用了循环模式，一直处于接收中断中，是否有时间发送数据呢？如果将发送功能改成中断方式发送数据会什么样？

7.8　小　　结

STM32F0 的 USART 功能是比较丰富的，除了可以 TTL 电平通信，另有常见的 RS-232、RS-485、RS-422，以及传感器领域使用 IO-Link 和低成本汽车总线-LIN 网等通信方式。STM32F0 为了简化 USART 软件部分工作量，提供了数据帧的硬件超时检测以及数据块尾检测功能。7.7 节演示了如何使用 STM32F0 的超时检测功能。

第8章

实时时钟（RTC）

实时时钟（RTC）是一个独立的 BCD 定时器/计数器。RTC 模块拥有一个具有可编程报警功能的时间日历时钟。RTC 模块拥有自动唤醒功能，用于管理所有的低功耗模式。两个 32 位寄存器以 BCD 格式存储秒、分、时（12/24h 制）、日（星期数）、日期、月和年。亚秒值同样可用二进制存储。RTC 具有自动月份、天数补偿功能，还包括夏令时间补偿。

单独使用一个 32 位寄存器存储可编程报警相关信息，包括亚秒、秒、分、时、日、日期。对由晶体本身的频偏、温度漂移及其他原因引起的任何误差，可以利用 RTC 本身的数字校准功能进行修正。上电复位后，所有 RTC 寄存器将被禁止访问，以防止意外的写操作。当设备处于运行模式、低功耗模式，或复位状态，只要电压在工作范围内，RTC 将保持正常运行。

STM32F0 的 RTC 相比以往几款 STM32F 系列芯片功能有所增加：精确到 RTCCLK 的亚秒级日历、入侵检测采用了双引脚/双事件带可配置滤波的电平检测。

在使用 LSE 时钟情况下（即 32.768kHz 外部晶振），RTC 的损耗 1μA。低损耗以及亚秒特性可将 RTC 应用于万年历、定时报警、时间的计时、计费等场合。

8.1 主 要 特 性

RTC 模块主要特性如下。

- ❑ 日历功能，可显示亚秒、秒、分、时（12/24 小时制）、日（星期）、日期、月和年。
- ❑ 通过软件编程实现夏令时补偿。
- ❑ 可编程报警中断功能。报警设定可由任何日历字段来触发。
- ❑ 参考时钟检测：采用更高精度的时钟源（50Hz 或 60Hz），用于扩展日历精度。
- ❑ 采用一个亚秒级外部时钟实现精确同步。
- ❑ 数字校准电路（计数器定期修正）：0.95×10^{-6} 精准度，获得一个几秒钟的校准窗口。
- ❑ 事件记录时间戳。
- ❑ 侵入检测事件：带可配置的滤波器和内部上拉电阻。
- ❑ 可屏蔽中断/事件：
  - ➢ Alarm A；
  - ➢ 时间戳；
  - ➢ 侵入检测。

- 备份寄存器：当产生一个侵入检测事件，备份寄存器被复位。

8.2　STM32F0 的 RTC 功能实现

表 8-1 是 STM32F0 系列芯片有关 RTC 功能的实现情况。

表 8-1　STM32F0 的 RTC 功能实现

RTC 特征	STM32F03x、STM32F04x、STM32F05x	STM32F07x
周期唤醒定时器	NA	X
RTC_TAMP1	X	X
RTC_TAMP2	X	X
RTC_TAMP3	NA	X
报警 A	X	X

8.3　功 能 描 述

8.3.1　RTC 框图

图 8-1 是 RTC 的框图。

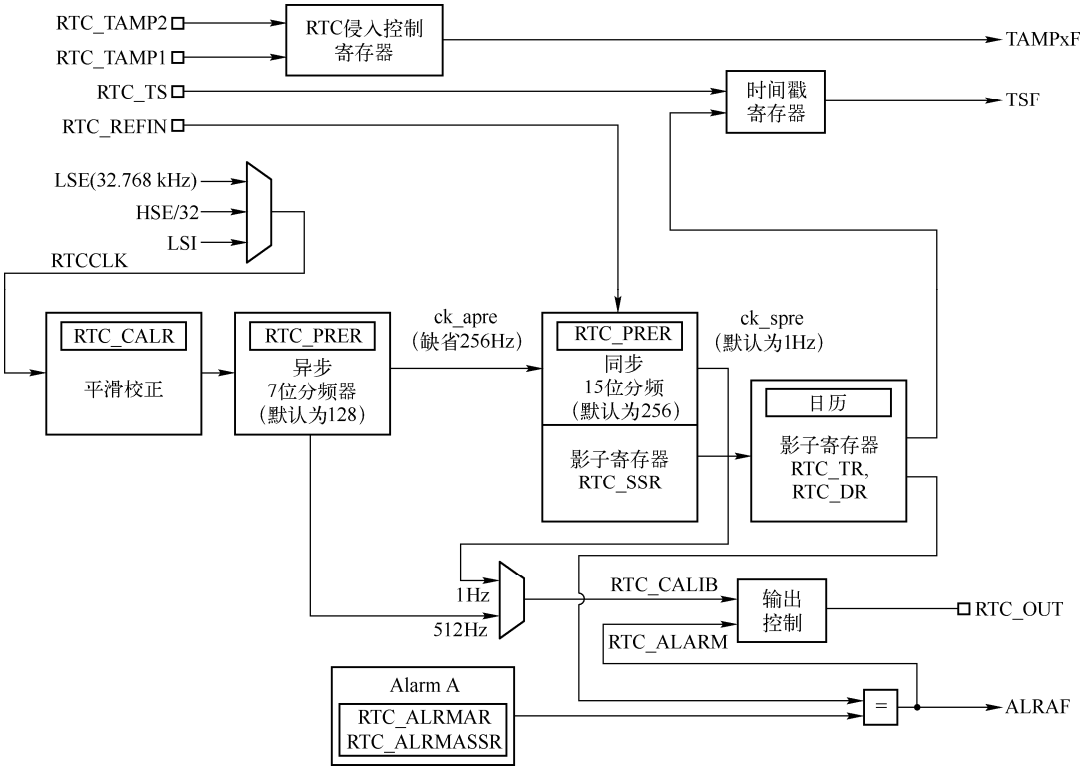


图 8-1　RTC 框图

注意：该图根据芯片有所不同。对于 LQFP32 以及 TSSOP20 封装的 STM32F0 芯片，已经无 Pc14 与 Pc15 引脚，所以也无 LSE 部分内容。

RTC 模块包括：

- 一个报警。
- 两个侵入事件。
- 复用功能输出，RTC_OUT 以下述两种形式中任意一种形式输出。
  - RTC_CALIB：512Hz 或 1Hz 时钟输出（用 32.768kHz 的 LSE 的时候），用 RTC_CR 寄存器的 COE 位开启。
  - RTC_ALARM：Alarm A。这个输出用 RTC_CR 寄存器中 OSEL[1:0]位来开启。
- 复用功能输入。
  - RTC_TS：时间戳事件。
  - RTC_TAMP1：侵入事件检测 1。
  - RTC_TAMP2：侵入事件检测 2。
  - RTC_REFIN：50Hz 或 60Hz 参考时钟输入。

8.3.2 被 RTC 控制的 GPIO

RTC_OUT、RTC_TS 和 RTC_TAMP1 映射到同一个引脚（PC13）。通过 RTC_TAFCR 寄存器完成 RTC_ALARM 输出的选择，其中 PC13VALUE 位用于选择 RTC_ALARM 输出是推挽模式或开漏模式。当 PC13 不用作 RTC 复用功能时，可通过设置 RTC_TAFCR 中的 PC13MODE 位将 PC14 强制为推挽输出模式，输出值由 PC13VALUE 位给出。此时，PC13 的推挽输出状态和值在待机模式下是可以保持的。

当 PC14 和 PC15 不用作 LSE 振荡器时，可通过分别设置 RTC_TAFCR 寄存器中的 PC14MODE 位和 PC15MODE 位将 PC14 和 PC15 强制为推挽输出模式，输出值由 PC14VALUE 和 PC15VALUE 给出。此时，PC14 和 PC15 的推挽输出状态和值在待机模式下是可保持的。

8.3.3 时钟和预分频器

LSE 振荡器、LSI 振荡器、HSE 振荡器均可作为 RTC 时钟。一个可编程预分频器产生一个 1Hz 的时钟，用于更新日历。为最大限度地减少功耗，预分频器可分割成两个可编程预分频器。

- 通过控制 RTC_PRER 寄存器中的 PREDIV_A 位来配置 7 位异步预分频器。
- 通过控制 RTC_PRER 寄存器中的 PREDIV_S 位来配置 15 位同步预分频器。

异步预分频器的分频系数设为 128，同步预分频器的分频系数设为 256，从而得到一个基于 32.768kHz LSE 频率的 1Hz（ck_spre）内部时钟频率。

8.3.4 实时时钟和日历

RTC 的日历时间寄存器和日期寄存器是通过影子寄存器访问的，如图 8-2 所示。该影子寄存器与 PCLK(APBclock)同步。为避免同步等待时间，也可直接访问。

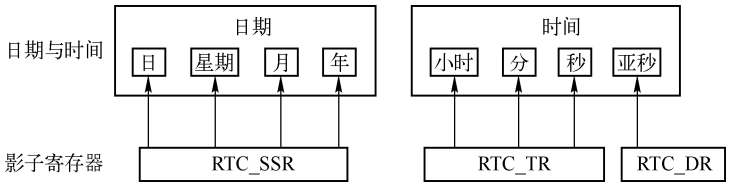


图 8-2　RTC 的影子寄存器与日历关系

硬件会间隔两个 RTCCLK 时钟周期更新一次影子寄存器的日历值，并将 RTC_ISR 寄存器的 RSF 位置 1。停止或待机模式下则不作这个更新，当退出上述两种模式后，影子寄存器将在最多两个 RTCCLK 周期后执行更新操作。

默认情况下，应用程序通过访问影子寄存器获取日历寄存器的内容。如需直接访问日历寄存器，可通过设置 RTC_CR 寄存器的 BYPSHAD 控制位（默认为零）来实现。

在 BYPSHAD=0 模式下，如需读取 RTC_SSR、RTC_TR 或 RTC_DR 寄存器的内容，APB 时钟频率（f_{APB}）至少是 RTC 时钟频率（f_{RTCCLK}）的 7 倍。系统复位后，影子寄存器也将自动复位。

8.3.5　可编程报警

RTC 模块拥有可编程报警功能：Alarm A。

设置 RTC_CR 寄存器的 ALRAE 位，启动可编程报警功能。当日历中亚秒、秒、分、时、日期或星期与报警寄存器 RTC_ALRMASSR 和 RTC_ALRMAR 中设定的值匹配，ALRAF 由硬件置 1。所有日历字段都可以通过 RTC_ALRMAR 寄存器的 MSKx 位和 RTC_ALRMASSR 寄存器的 MASKSSx 位独立地被选择为报警源。设置 RTC_CR 寄存器的 ALRAIE 位，会产生报警中断。

设置 RTC_CR 寄存器的 OSEL[0:1]启动 Alarm A，可同步输出至 RTC_ALARM。RTC_ALARM 输出极性可由 RTC_CR 寄存器的 POL 位设置。

8.3.6　RTC 初始化及配置

1．访问 RTC 寄存器

RTC 寄存器是 32 位寄存器。除了 BYPSHAD=0 时对日历寄存器执行读操作外，APB 接口为其他对 RTC 寄存器的访问提供了两种等待状态。

2．RTC 寄存器的写保护

上电复位后，所有 RTC 寄存器将处于写保护状态。通过向写保护寄存器 RTC_WPR 写入指定关键字来启动 RTC 寄存器的写权限。通过以下操作解除所有 RTC 寄存器（RTC_ISR[13:8],RTC_TAFCR 和 RTC_BKPxR 除外）的写保护。

（1）向 RTC_WPR 寄存器写入“0xCA”。

（2）向 RTC_WPR 寄存器写入“0x53”。如果写入错误将重新激活写保护。系统复位不影响写保护机制。

3．日历初始化及配置

按照以下顺序完成时间和日期值的初始化，包括时间格式和预分频器的配置。

（1）RTC_ISR 寄存器的 INIT 位置 1，进入初始初始化模式。在该模式下日历计数器暂停运行，此时可更新计数器的值。

（2）等待 RTC_ISR 寄存器的 INITF 位置 1，确保已经正式进入初始化模式。由于时钟同步的延迟，该过程大约需要两个 RTCCLK 时钟周期。

（3）为了使日历计数器生成一个 1Hz 的时钟，编写 RTC_PRER 寄存器中的分频系数。

（4）将初始时间和日期值加载到影子寄存器（RTC_TR 和 RTC_DR），通过 RTC_CR 寄存器的 FMT 位设置时间格式（12h 制/24h 制）。

（5）清除 INIT 位的值退出初始化模式。日历计数器的实际值将会自动加载，并在 4 个 RTCCLK 时钟周期后重新启动。

在完成上述一系列初始化操作后，日历将开始计时。

4．夏令时

通过 RTC_CR 寄存器的 SUB1H、ADD1H 和 BKP 位管理夏令时间。通过设置 SUB1H 或 ADD1H 位，软件可在不通过初始化流程的情况下将日历中的时间单次增加/减少 1h。此外，软件可通过 BKP 位记录该操作。

5．可编程报警

编程或更新可编程报警时，应遵循以下几点：

- 清除 RTC_CR 的 ALRAE 位禁用 Alarm A；
- 编程 Alarm A 寄存器（RTC_ALRMASSR/RTC_ALRMAR）；
- 设置 RTC_CR 的 ALRAE 位重新启用 Alarm A。

8.3.7 读日历寄存器

1．当 RTC_CR 寄存器的 BYPSHAD 控制位被清除时

为确保在安全同步机制下正常读 RTC 日历寄存器（RTC_SSR、RTC_TR 和 RTC_DR），APB1 时钟频率（f_{PCLK}）应至少为 RTC 时钟频率（f_{RTCCLK}）的 7 倍。

当 APB1 时钟频率低于 7 倍 RTC 时钟频率时，软件必须两次读取日历时间和日期寄存器。如果第二次读取 RTC_TR 返回的值与第一次返回的值相同，说明返回值是正确的，否则需再次读取。任何情况下，APB1 时钟频率都必须大于 RTC 时钟频率。

日历寄存器的内容被复制到 RTC_TR 和 RTC_DR 影子寄存器中时，RTC_ISR 寄存器的 RSF 位被置位。复制操作每两个 RTCCLK 周期执行一次。为确保三者的值保持一致，读 RTC_SSR 或 RTC_TR，锁定高阶日历影子寄存器中的值，直至 RTC_DR 中的值被读取。为避免软件在时间间隔少于两个 RTCCLK 周期的情况下多次访问日历，每次读日历 RSF 位应由软件清零，软件必须等待 RSF 位被置位后才能读 RTC_SSR、RTC_TR 和 RTC_DR 寄存器。

从低功耗模式（关机或待机）唤醒后，RSF 位应由软件清零，软件必须等待 RSF 位被再次置位后才能读 RTC_SSR、RTC_TR 和 RTC_DR 寄存器。RSF 位应在唤醒后被清除，而不是进入低功耗模式前。

系统复位后，软件必须等待 RSF 位被置位后才能读 RTC_SSR、RTC_TR 和 RTC_DR 寄存器。事实上系统复位将导致影子寄存器复位至其默认值。初始化后，软件必须等待 RSF 位被置位后才能读 RTC_SSR、RTC_TR 和 RTC_DR 寄存器。同步（见 8.3.8 节）后，软件必

须等待 RSF 位被置位后才能读 RTC_SSR、RTC_TR 和 RTC_DR 寄存器。

2．当 RTC_CR 寄存器的 BYPSHAD 控制位被置位时（无须考虑影子寄存器）

读日历寄存器，直接从日历计数器获取值，无须等待 RSF 位被置位。此功能在刚退出低功耗模式（停止或待机模式）时非常有用，因为影子寄存器在低功耗模式下不会自动更新。

在 BYPSHAD 为 1 时，如果两次读寄存器之间出现 RTCCLK 边沿，不同寄存器中的结果可能会互不相关。此外，如果在读操作过程中遇到 RTCCLK 边沿，则某个寄存器的值可能不正确。软件必须读取所有的寄存器两次，并比较两次读取的结果，或者通过比较两组最低有效日历寄存器的结果，以检验数据是否正确且有一定关联。

8.3.8　复位过程

任何可用的系统复位源都将导致日历影子寄存器（RTC_SSR、RTC_TR 和 RTC_DR）和 RTC 状态寄存器（RTC_ISR）复位至默认值。

然而，下列寄存器的复位与系统复位无任何关联，只与上电复位有关：RTC 当前日历寄存器、RTC 控制寄存器（RTC_CR）、预分频器寄存器（RTC_PRER）、RTC 校准寄存器（RTC_ CALR）、RTC 移位寄存器（RTC_SHIFTR）、RTC 时间戳寄存器（RTC_TSSSR、RTC_TSTR 和 RTC_TSDR）、RTC 侵入和复用功能配置寄存器（RTC_TAFCR）、RTC 备份寄存器（RTC_BKPxR）、Alarm A 寄存器（RTC_ALRMASSR/RTC_ALRMAR）。

除上电复位外，任何系统复位时 RTC 将维持运行状态。发生上电复位后，RTC 停止运行，所有 RTC 寄存器复位至默认值。

8.3.9　RTC 同步

RTC 可与一个高精度远程时钟同步。在读取亚秒字段（RTC_SSR 或 RTC_TSSSR）后，可计算出远程时钟和 RTC 中时间的精确偏差值。RTC 可以通过 RTC_SHIFTR 寄存器用时钟移位的方式瞬间剔除这个偏差。

RTC_SSR 用于存储同步预分频器的计数器值。这允许用 1/(PREDIV_S+1)秒的分辨率去衡量 RTC 所保持的实际时间。因此，这个分辨率可以通过增加同步预分频值（PREDIV_S[14:0]）的方式提高。当 PREDIV_S 被设为 0x7FFF 时可得到允许的最大分辨率为 32768Hz 时钟时的 30.52μs。

然而，如果 PREDIV_S 增加，为了保证同步预分频器的输出为 1Hz，PREDIV_A 必须减少。在该情况下，同步预分频器的输出频率将增加，同时动态功耗也会增加。

通过 RTC 移位控制寄存器（RTC_SHIFTR）可以微调 RTC。写 RTC_SHIFTR 寄存器可能移位（延迟或者提前）时钟信号，在分辨率为 1/(PREDIV_S+1)秒的时候幅度可达 1s。移位操作包括将 SUBFS[14:0]的值加到同步预分频计数器 SS[15:0]，这将延迟时钟信号。如果同时 ADD1S 位是 1，结果会是加上一个整秒同时再减去不到 1s，所以这会提前时钟信号。

写 RTC_SHIFTR 寄存器启动移位操作后，通过硬件设置 SHPF 标志，表示一个移位操作正在等待。当移位操作完成后该位被硬件清除。

8.3.10　RTC 参考时钟检测

RTC_REFIN 参考时钟（50Hz 或 60Hz）相比 32.768kHz LSE 时钟，具有更高的精确

度。RTC_REFIN 检测启用后（RTC_CR 的 REFCKON 位置 1），将用于补偿日历更新频率（1Hz）的偏移。图 8-3 是利用市电进行参考时钟检测的框图。

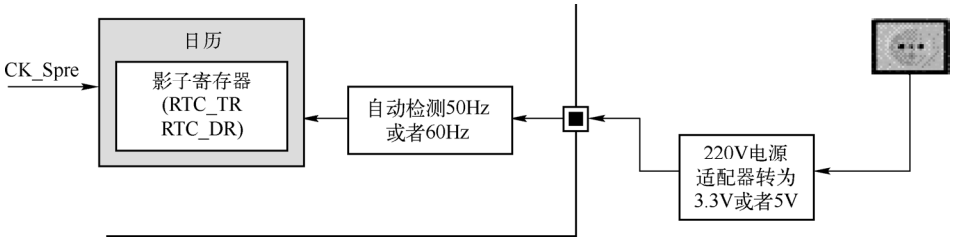


图 8-3　参考时钟检测

每个 1Hz 的时钟沿会和最近的 RTC_REFIN 时钟沿（如果在给定的时间窗口中可以找到）进行比较。多数情况下两个时钟沿会恰好对齐。当发现对得不齐，说明 LSE 不精确，RTC 将 1Hz 时钟移动一个最小位，从而下一个沿就就能对齐。得益于这种机制使得这个万年历的精度变得和参考时钟一样。

当参考时钟停止时，日历将完全遵照 LSE 时钟进行更新。RTC 等待参考时钟，并产生一个围绕 Ck_Spre 边沿的检测窗口。

启动 RTC_REFIN 检测后，PREDIV_A 和 PREDIV_S 必须复位至默认值：

PREDIV_A = 0x007F

PREVID_S = 0x00FF

8.3.11　RTC 平滑数字校准

RTC 频率可以按大约 0.954×10^{-6} 的分辨率进行校准，范围为-487.1×10^{-6}～$+488.5\times10^{-6}$。通过一系列微调（如增加/减少单独的 RTCCLK 脉冲）进行频率校正。这些校准会分布得相当好，所以 RTC 会校准得很好，即便是再观察一段时间也一样。在一个约 2^{20} RTCCLK 脉冲周期或 32 秒（输入频率为 32768Hz 时）内，执行平滑数字校准。

8.3.12　时间戳功能

RTC_CR 寄存器 TSE 位置 1 启用时间戳功能。当 RTC_TS 引脚检测到一个时间戳事件时候，会将日历存储在时间戳寄存器中（RTC_TSSSR、RTC_TSTR、RTC_TSDR）。当产生时间戳事件时，RTC_ISR 寄存器的时间戳标志位（TSF）被置位。通过设置 RTC_CR 寄存器的 TSIE 位，在产生时间戳事件时，同时产生一个中断。如果检测到一个新的时间戳事件时，时间戳标志位（TSF）已经被置位，则时间戳溢出标志位（TSOVF）标志为“溢出”，时间戳寄存器（RTC_TSTR 和 RTC_TSDR）维持上一事件的内容不变。

8.3.13　侵入检测

RTC_TAMPx 输入事件可设置为边沿检测或带过滤功能的电平检测。侵入检测可被配置成：

- ❑ 擦除 RTC 备份域寄存器；
- ❑ 停止和待机模式的唤醒中断产生。

1．RTC 备份寄存器

备份寄存器（RTC_BKPxR）处在 VDD 备份域中，当 VDD 电源被切断，它们仍由 VBAT 维持供电。当系统复位或设备在待机模式下被唤醒时，它们不会被复位。上电复位时，备份寄存器将被复位。当产生一个侵入检测事件时，备份寄存器将被复位。

2．侵入检测初始化

所有 RTC_TAMPx 侵入检测输入均与 RTC_ISR2 寄存器的 TAMPxF 标志相关。所有输入可通过设置 RTC_TAFCR 寄存器中相应的 TAMPxE 位为“1”来使能。侵入检测事件可将所有备份寄存器（RTC_BKPxR）内容清除。通过设置 RTC_TAFCR 寄存器的 TAMPIE 位，当检测到一个侵入检测事件时便产生一个中断。

3．侵入事件时间戳

设置 TAMPTS 为 1，所有侵入事件将产生一个时间戳。这种情况下，RTC_ISR 中无论是 TSF 位还是 TSOVF 位被置 1，和正常的时间戳事件发生的效果相同。设置 TSF 或 TSOVF，与其关联的侵入标志寄存器 TAMPxF 将同时被设置。

4．侵入输入的边沿检测

如果 TAMPFLT 位为 00，根据相关 TAMPxTRG 位，当检测到上升沿或下降沿时，RTC_TAMPx 引脚将生成侵入检测事件。当边沿检测被选中后，RTC_TAMPx 输入的内部上拉电阻将被停用。

- 当 TAMPxTRG= 0：如果在启用侵入检测（通过设置 TAMPxE 位为 1）前 RTC_TAMPx 已经为高电平，一旦启用 RTC_TAMPx 输入，则会产生一个侵入事件（尽管在 TAMPxE 位置“1”后 RTC_TAMPx 输入中并没有出现上升沿）。
- 当 TAMPxTRG=1：如果在启用侵入检测前，RTC_TAMPx 已经为低电平，一旦启用 RTC_TAMPx，则会产生一个侵入事件（尽管在设置 TAMPxE 位后 RTC_TAMPx 输入中并没有出现下降沿）。

在一个侵入事件被检测到并清除后，RTC_TAMPx 复用功能应该被禁止。然后在再次写入备份寄存器前重新启用 RTC_TAMPx 复用功能（通过设置 TAMPxE 位为 1），这样可以阻止软件在 RTC_TAMPx 检测出仍有侵入事件发生时对备份寄存器进行写操作。这相当于对 RTC_TAMPx 复用功能输入进行电平检测。

5．针对 RTC_TAMPx 输入的带滤波功能电平检测

将 TAMPFLT 设置为非“0”值，会启用带滤波功能的电平检测。当检测到 2 个、4 个或 8 个（根据 TAMPFLT 设置）连续样本的指定电平（TAMPxTRG 位决定）时，将产生一个侵入检测事件。

RTC_TAMPx 输入在其状态被采样前，通过 I/O 内部上拉电阻实现预充电，直至 TAMPPUDIS 设置为“1”后，停止该功能。预充电时间由 TAMPPRCH 位决定，允许 RTC_TAMPx 输入拥有更大的电容。可通过调整 TAMPFREQ 来改变电平检测的采样频率，从而在侵入检测延迟与通过上拉电阻产生的电源消耗间进行取舍。

8.3.14　校准时钟输出

当 RTC_CR 寄存器 COE 位置“1”，RTC_CALIB 输出上会提供一个参考时钟。如果 RTC_CR 寄存器 COSEL 位为 0，且 PREDIV_A=0x7F，RTC_CALIB 频率为 $f_{RTCCLK}/64$，对应

于 32.768kHz 的 RTCCLK 时，输出频率为 512Hz。因为下降沿存在轻微抖动，所以 RTC_CALIB 占空比是不规则的，因此建议使用上升沿。

如果 COSEL=1，“PREDIV_S+1”为非“0”的 256 的倍数（例如 PREDIV_S[7:0]= 0xFF），RTC_CALIB 频率等于 fRTCCLK/(256*(PREDIV_A+1))。这样在 RTCCLK 频率为 32.768kHz 的时候，对于的 RTC_CALIB 输出频率为 1Hz（默认 PREDIV_A= Ox7F，PREDIV_S=0xFF）。

8.3.15 报警输出

RTC_CR 寄存器的 OSEL[1:0]控制位的功能是激活报警备用功能 RTC_ALARM 输出，并选择输出功能。这些功能与 RTC_ISR 寄存器中相应标志位的内容一致。输出极性由 RTC_CR 寄存器的 POL 控制位决定，因此当 POL 设置为“1”时，就会输出选定标志位的取反后的值。

通过设置 RTC_TAFCR 寄存器 ALARMOUTTYPE 控制位，RTC_ALARM 引脚可被设置为开漏输出或推挽输出。

8.4 RTC 低功耗模式

表 8-2 是 RTC 的低功耗模式。

表 8-2 RTC 低功耗模式

模 式	描 述
睡眠模式	无影响 RTC 中断使设备退出睡眠模式
停止模式	如果 RTC 时钟源为 LSE 或 LSI，RTC 将保持运行状态。RTC 报警、RTC 侵入事件、RTC 时间戳事件和 RTC 唤醒事件使设备退出停止模式
待机模式	如果 RTC 时钟源为 LSE 或 LSI，RTC 将保持运行状态。RTC 报警、RTC 侵入事件、RTC 时间戳事件和 RTC 唤醒事件使设备退出待机模式

8.5 RTC 中断

所有 RTC 中断都连接到 EXTI 控制器。EXTI 17 连接 RTC 报警事件；EXTI 19 连接 RTC 侵入和时间戳事件；EXTI 20 连接 RTC 唤醒事件（只用于 STM32F07X）。表 8-3 是 RTC 的中断控制位以及中断事件的影响。

1．启用 RTC 报警中断的过程

（1）在中断模式下设置并使能 RTC 报警事件对应的 EXTI 线，并选定上升沿极性。

（2）设置并使能 NVIC 的 RTC_ALARM IRQ 通道。

（3）设置 RTC，触发 RTC 报警（Alarm A）。

2．启用侵入中断的顺序

（1）在中断模式下设置并使能 RTC 侵入事件对应的 EXTI 线，并选定上升沿极性。

（2）设置并使能 NVIC 的 TAMP_STAMP IRQ 通道。

（3）设置 RTC，检测 RTC 侵入事件。

3．启用 RTC 时间戳中断的顺序

（1）在中断模式下设置并使能 RTC 时间戳事件对应的 EXTI 线，并选定上升沿极性。

（2）设置并使能 NVIC 的 TAMP_STAMP IRQ 通道。

（3）设置 RTC，检测 RTC 时间戳事件。

表 8-3　中断控制位

中断事件	事件标志	使能控制位	退出睡眠模式	退出停止模式
Alarm A	ALRAF	ALRAIE	是	是
RTC_TS 输入（时间戳）	TSF	TSIE	是	是
RTC_TAMP1 输入检测	TAMP1F	TAMPIE	是	是
RTC_TAMP2 输入检测	TAMP2F	TAMPIE	是	是

8.6　固　件　库

RTC 的固件库根据 RTC 功能分成了 14 组，见表 8-4。

表 8-4　RTC 的固件库

组	函　数　名	描　　述
设置默认状态	RTC_DeInit	恢复 RTC 寄存器为默认复位值
初始化和配置	RTC_Init	由函数的输入参数初始化 RTC 寄存器
	RTC_StructInit	使用默认值填充 RTC_InitStruct 变量
	RTC_RefClockCmd	使能或禁用 RTC 参考时钟检测
	RTC_EnterInitMode	进入 RTC 初始化模式
	RTC_ExitInitMode	退出 RTC 初始化模式
	RTC_WriteProtectionCmd	使能或退出写保护
	RTC_WaitForSynchro	等待 RTC 时间和日期寄存器（RTC_TR 和 RTC_DR）被同步
	RTC_TimeStructInit	默认值设置 RTC_TimeStruct 成员变量
	RTC_BypassShadowCmd	使能或禁用旁路影子特性
RTC 时钟、日期函数	RTC_SetTime	设置 RTC 当前时间（小时、分、秒、12h 时钟周期（AM/PM））
	RTC_SetDate	设置 RTC 当前日期
	RTC_GetTime	读取 RTC 当前时间
	RTC_GetDate	读取 RTC 当前日期
	RTC_DateStructInit	默认值设置 RTC_DateStruct
	RTC_TimeStructInit	默认值设置 RTC_TimeStruct
	RTC_GetSubSecond	读取 RTC 当前日历亚秒值
RTC 报警	RTC_SetAlarm	设置 RTC 报警配置：报警时间、报警掩码、报警日期/星期选择、报警日期/星期值
	RTC_GetAlarm	获取 RTC 指定的报警配置
	RTC_AlarmCmd	使能或指定 RTC 特定报警

续表

组	函 数 名	描 述
RTC 报警	RTC_AlarmStructInit	默认值填充 RTC_AlarmStruct 成员
	RTC_AlarmSubSecond Config	配置 RTC 报警 A/B 亚秒值和掩码
	RTC_GetAlarmSubSecond	获取 RTC 报警亚秒值
RTC 唤醒	RTC_WakeUpClockConfig	配置 RTC 唤醒时钟源
	RTC_SetWakeUpCounter	设置 RTC 唤醒计数器值
	RTC_GetWakeUpCounter	返回 RTC 唤醒定时器计数器值
	RTC_WakeUpCmd	使能或禁用 RTC 唤醒定时器
夏令时	RTC_DayLightSavingConfig	根据夏令时参数加减 1 小时
	RTC_GetStoreOperation	返回夏令时存储操作
输出引脚配置	RTC_OutputConfig	配置 RTC 输出引脚
数字平滑校准	RTC_CalibOutputCmd	使能或禁用 RTC 输出
	RTC_CalibOutputConfig	配置校准引脚（RTC_CALIB）选择（1Hz 或 512Hz）
	RTC_SmoothCalibConfig	配置平滑校准设置
时间戳	RTC_TimeStampCmd	使能或禁用 RTC 时间戳功能
	RTC_GetTimeStamp	读取 RTC 时间戳值和掩码
	RTC_GetTimeStampSubSecond	读取 RTC 时间戳亚秒值
侵入函数	RTC_TamperTriggerConfig	配置侵入沿触发器
	RTC_TamperCmd	使能或禁用侵入检测
	RTC_TamperFilterConfig	配置侵入滤波
	RTC_TamperSamplingFreqConfig	配置侵入采样频率
	RTC_TamperPinsPrechargeDuration	配置侵入引脚输入预充电时间
	RTC_TimeStampOnTamperDetectionCmd	使能或禁用侵入引脚
	RTC_TamperPullUpCmd	使能或禁用在侵入检测事件上的时间戳
后备域寄存器	RTC_WriteBackupRegister	往 RTC 后备域数据寄存器写数据
	RTC_ReadBackupRegister	从 RTC 后备域数据寄存器读取数据
时间戳、侵入引脚配置	RTC_OutputTypeConfig	配置 RTC 输出引脚模式（开漏/上下拉）
移位同步控制	RTC_SynchroShiftConfig	配置同步移位控制
	RTC_ITConfig	使能或禁用特定的 RTC 中断
标志和中断函数	RTC_GetFlagStatus	检查特定的 RTC 标志是否被设置
	RTC_ClearFlag	清除 RTC 挂起标志
	RTC_GetITStatus	检查特定 RTC 中断是否发生
	RTC_ClearITPendingBit	清除 RTC 中断挂起标志位

1．固件库用法

- 使能后备域的访问。复位后，后备域（RTC 寄存器和 RTC 后备域数据）处于写保护状态，为了使能，需要使用 RCC_APB1PeriphClockCmd()函数使能 PWR APB1 接口，使用 PWR_BackupAccessCmd()函数使能后备域，使用 RCC_RTCCLKConfig()选择 RTC 时钟源，使用 RCC_RTCCLKCmd()使能 RTC 时钟。
- 使用 RTC_Init()函数配置 RTC 的预分频（同步和异步）和 RTC 小时格式。

2．时间和日期配置

- 使用 RTC_SetTime()函数和 RTC_SetDate()函数设置 RTC 日历（时间和日期）。
- 通过 RTC_GetTime()函数和 RTC_GetDate()读取 RTC 日历。
- 通过 RTC_GetSubSecond()函数读取 RTC 亚秒。
- 使用 RTC_DayLightSavingConfig()函数加减 1 小时。

3．报警设置

- 使用 RTC_SetAlarm()函数配置 RTC 报警。
- 使用 RTC_AlarmCmd()函数使能或禁用 RTC 报警。
- 通过 RTC_GetAlarm()函数读取 RTC 报警。
- 使用 RTC_GetAlarmSubSecond()函数读取 RTC 报警亚秒。

4．RTC 唤醒配置

- 使用 RTC_WakeUpClockConfig()函数配置 RTC 唤醒时钟源。
- 使用 RTC_SetWakeUpCounter()函数配置 RTC 唤醒计数器。
- 使用 RTC_WakeUpCmd()函数使能 RTC 唤醒。
- 通过 RTC_GetWakeUpCounter()函数读取 RTC 唤醒计数器寄存器。

5．输出配置

- 用于管理 RTC Alarm A 的 AFO_ALARM 输出，使用 RTC_OutputConfig()函数选择 RTC 信号输出引脚。
- 输出 512Hz 或者 1Hz 信号的 AFO_CALIB，通过 RTC_CalibOutputCmd()设置。

6．时间戳配置

- 使用 RTC_TimeStampCmd()函数配置 RTC_AF1 触发器，使能 RTC 时间戳。
- 通过 RTC_GetTimeStamp()函数读取 RTC 时间戳时间和日期寄存器。
- 通过 RTCTimeStampSubSecond()函数读取 RTC 时间戳亚秒寄存器。

8.7　闹钟报警实例

RTC 具有时钟日历功能，并具有报警功能，可使用 RTC 搭建一个闹钟报警功能，用于定时提醒。

由于 RTC 的中断是连接到 EXTI 17 上，需进行 EXTI 的配置。由于 RTC 的功能是体现不间断运行，对于可采用电池供电的 STM32F051 以及 STM32F072 之类的芯片，可使用 LSI 低功耗时钟。如果仅提供 RTC 功能，功耗比较低，可采用电池供电。例如，计算机的 CMOS 通过电池供电，一般计算机在使用很久之后才会出现 CMOS 的日期丢失现象，即电池没电现象。采用电池供电的 STM32F0 芯片，也会发生这现象，需要更换电池重新设置日期。

对于 STM32F030F4 芯片，由于无电池功能。RTC 的时钟源只能是 HSE 或者 LSI，需要在每次使用时进行重新设置日历。

下面代码选用 LSI 作为 RTC 的时钟源，通过 RTC_Config()设置 RTC 时钟源，RTC_TimeRegulate()函数设置日历以及报警设置。在 main 函数中设置 RTC 的 Alarm A 以

及 EXTI17，使能 RTC 中断。RTC_TimeShow()与 RTC_AlarmShow()是读取时间值与报警设置值。

```
int main(void)
{ RTC_InitTypeDef RTC_InitStructure;
  NVIC_InitTypeDef   NVIC_InitStructure;
  EXTI_InitTypeDef   EXTI_InitStructure;
  USART_Configuration();
 if (RTC_ReadBackupRegister(RTC_BKP_DR0) != BKP_VALUE)//未配置时钟信息，STM32F030F4不支持该功能。
  {
    /*RTC 配置   */
    RTC_Config();
        /*分频设置 */
    RTC_InitStructure.RTC_AsynchPrediv = AsynchPrediv;
    RTC_InitStructure.RTC_SynchPrediv = SynchPrediv;
    RTC_InitStructure.RTC_HourFormat = RTC_HourFormat_24;
        /* 检查是否已经初始化 */
    if (RTC_Init(&RTC_InitStructure) = = ERROR)
    {
      printf("\n\r          /!\\***** RTC Prescaler Config failed ********/!\\ \n\r");
    }

    /* 配置时间寄存器 */
    RTC_TimeRegulate();
  }
  else //已经配置过时钟信息，无须配置时钟信息
  {
    /* 使能 PWR 时钟 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);
    /* RTC 访问使能   */
    PWR_BackupAccessCmd(ENABLE);
   /* 使能 LSI OSC */
    RCC_LSICmd(ENABLE);
    /*等待 RTC APB 寄存器同步*/
    RTC_WaitForSynchro();
    /* 清除 RTC 报警标志 */
    RTC_ClearFlag(RTC_FLAG_ALRAF);
    /* 清除 EXTI 17 挂起标志（内部连接 RTC 报警）*/
    EXTI_ClearITPendingBit(EXTI_Line17);
    /* 显示 RTC 时间和报警 */
    RTC_TimeShow();
    RTC_AlarmShow();
  }
  /* RTC Alarm A 中断配置 */
  /* EXTI 配置*/
```

```
    EXTI_ClearITPendingBit(EXTI_Line17);
    EXTI_InitStructure.EXTI_Line = EXTI_Line17;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);
    /*  使能 RTC 报警中断  */
    NVIC_InitStructure.NVIC_IRQChannel = RTC_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    /* Infinite loop */
    while (1)
    {
    }
}

/*配置 RTC*/
static void RTC_Config(void)
{     /*  使能 PWR 时钟  */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);
    /*  允许访问 RTC */
    PWR_BackupAccessCmd(ENABLE);
  /*  当使用 LSI 作为 RTC 时钟源*/
/* The RTC Clock may varies due to LSI frequency dispersion. */
    /*  使能  LSI 振荡  */
    RCC_LSICmd(ENABLE);
    /*  等待到 LSI 预备*/
    while(RCC_GetFlagStatus(RCC_FLAG_LSIRDY) == RESET)
    {
    }
    /*  把 RTC 时钟源配置为 LSI */
    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSI);
     /*  同步分频值和异步分频值  */
    SynchPrediv = 0x18F;
    AsynchPrediv = 0x63;
    /*  使能 RTC 时钟  */
    RCC_RTCCLKCmd(ENABLE);
    /*  等待 RTC APB 寄存器同步  */
    RTC_WaitForSynchro();
}
/*  设置当前时间以及报警设置，开启 RTC 中断*/
void RTC_TimeRegulate(void)
{   RTC_TimeTypeDef RTC_TimeStructure;
    RTC_AlarmTypeDef    RTC_AlarmStructure;
      RTC_TimeStructure.RTC_H12         = RTC_H12_AM;
```

```
    RTC_TimeStructure.RTC_Hours=10;
    RTC_TimeStructure.RTC_Minutes=0;
    RTC_TimeStructure.RTC_Seconds=0;
  /*配置 RTC 时间寄存器*/
  if(RTC_SetTime(RTC_Format_BIN, &RTC_TimeStructure) == ERROR)
  {
      printf("\n\r>> !! RTC Set Time failed. !! <<\n\r");
  }
  else
  {
     RTC_TimeShow();
    //   RTC_WriteBackupRegister(RTC_BKP_DR0, BKP_VALUE);//由于 STM32F030F4 无 Vbat，
此功能不起作用
  }
  /*禁用 Alarm A */
  RTC_AlarmCmd(RTC_Alarm_A, DISABLE);
  RTC_AlarmStructure.RTC_AlarmTime.RTC_H12 = RTC_H12_AM;
    RTC_AlarmStructure.RTC_AlarmTime.RTC_Hours=10;
    RTC_AlarmStructure.RTC_AlarmTime.RTC_Minutes=1;
    RTC_AlarmStructure.RTC_AlarmTime.RTC_Seconds=0;
    /*  设置 Alarm A */
  RTC_AlarmStructure.RTC_AlarmDateWeekDaySel = RTC_AlarmDateWeekDaySel_Date;
  RTC_AlarmStructure.RTC_AlarmDateWeekDay = RTC_Weekday_Monday;
  RTC_AlarmStructure.RTC_AlarmMask = RTC_AlarmMask_DateWeekDay;
  /*  配置 RTC 报警 A 寄存器*/
  RTC_SetAlarm(RTC_Format_BIN, RTC_Alarm_A, &RTC_AlarmStructure);
  RTC_AlarmShow();
  /*  使能 RTC Alarm A 中断  */
  RTC_ITConfig(RTC_IT_ALRA, ENABLE);
  /*使能 alarm A */
  RTC_AlarmCmd(RTC_Alarm_A, ENABLE);
}
RTC 的中断函数在处理完中断后，需要清除 Alarm A 与 EXTI 17 的中断标志。
void RTC_IRQHandler(void)
{
  if(RTC_GetITStatus(RTC_IT_ALRA) != RESET)
  {
          LED_Toggle();
          RTC_AlarmShow();
          RTC_TimeShow();
    RTC_ClearITPendingBit(RTC_IT_ALRA);
    EXTI_ClearITPendingBit(EXTI_Line17);
  }
}
```

8.8　小　　结

本章是关于 RTC 的内容，主要有 RTC 的时钟源、后备域、报警、侵入以及 RTC 的固件库等内容。RTC 的功能主要是体现在时钟日历功能上，以及由此延伸出来报警以及日历保护上。但日历的功能基础是不掉电，所以 RTC 的时钟源与供电选择有关。

第 9 章

看 门 狗

微控器很容易受到外界的干扰，如电源电压波动、电机起停，受到干扰后程序可能跑飞、不按原来的逻辑顺序执行、跳到另一地方执行，或者陷入一个死循环。启用看门狗后，如果在设定的时间内没有执行重置计数值指令，微控器将复位，重新开始执行程序，保证系统恢复运行。

看门狗定时器（Watch Dog Timer，WDT）主要由一个专用的定时器组成，当启动看门狗后，定时器开始向下计数，计数为 0 时将产生一个中断，复位微控器。看门狗定时器和 PWM 定时功能虽然都是利用计数功能，但目的不一样。看门狗的特点是，需要不停地对其提供喂狗信号（一些外置看门狗芯片）或重新设置计数值，保持计数值不为 0。一旦计数值为 0，看门狗将发出复位信号复位系统或产生中断。

9.1 STM32F0 看门狗概述

STM32F0 有两个看门狗，分别是独立看门狗（Independent WatchDog，IWDG）和窗口看门狗（system Window WatchDog，WWDG）。

独立看门狗由专用的 40kHz 的低速时钟驱动，即使主时钟发生故障它也仍然有效。IWDG 也能够工作在窗口看门狗模式下；窗口看门狗由从 APB1 时钟分频后得到时钟驱动，通过可配置的时间窗口来检测应用程序非正常的过迟或过早的操作。

IWDG 适合应用于那些需要看门狗在主程序之外，能够完全独立工作，并且对时间精度要求较低的场合。WWDG 适合那些要求看门狗在精确计时窗口起作用的应用程序。

表 9-1 是 IWDG 和 WWDG 之间的对比情况。

表 9-1　独立看门狗和窗口看门狗对比

对比项	独立窗口看门狗	窗口看门狗
时钟源	LSI（40kHz）	PCLK（max=48MHz）
自减计数器	LSI /4(8,16,…,256)	PCLK/[4896*1(2,4,8)]
定时范围	[100μs, 26.2s]	[85.33μs, 43.69ms]
产生系统复位	计数器自减到 0，喂狗时间在窗外	计数器自减到 0x39，喂狗时间在窗外
产生中断		计数器自减到 0x40 产生 EWI

续表

对比项		独立窗口看门狗	窗口看门狗
喂狗	时间窗	设定的窗口值 >计数器的值 >0	设定的窗口值 >计数器的值 >0x39
	方式	0xAAAA　->IWWDG_KR	写计数器@WWDG_CR.T[6:0]
开启	软件	0xCCCC　->IWWDG_KR	置位 WWDG_CR.WDGA
	硬件	选项字节 WDG_SW	
窗口寄存器		WIN[11:0]复位值=0xFFF	W[6:0]复位值=0x7F
注意事项		低功耗模式下仍可运行； 一旦使能硬件看门狗，系统上电后自动运行； 一旦启动了看门狗，就不能关闭它，也不能关闭 LSI 了	复位后总是禁止，一旦通过 WDGA 来启动，除非复位不能再关掉它； 使能时 T6@CR 也要为 1，以避免立刻复位； 可以利用 T6 来产生软件复位； 使用前需开启 WWDG 的门控时钟

因为独立看门狗的不可停止特点，使得独立看门狗与待机睡眠的低功耗在使用上有些冲突。为了防止复位的发生，可通过 RTC 定时唤醒喂狗的方式，或者在进入低功耗模式时，首先执行一次软件复位，复位后重启关闭看门狗，而后再进入低功耗模式。

9.2　独立看门狗（IWDG）

STM32F0 器件集成了一个内嵌的看门狗外设，用来解决某些软件故障问题的。当它的定时计数值达到了预设的门限时，它会触发一个系统复位请求。独立看门狗由它自己专有的低速时钟来驱动，因此主时钟失效了，它仍然能保持工作状态。该外设非常适合在主程序以外需要一个完全独立看门狗，但对时钟精确度却又不太高的应用。

图 9-1 描述了独立看门狗模块的功能。

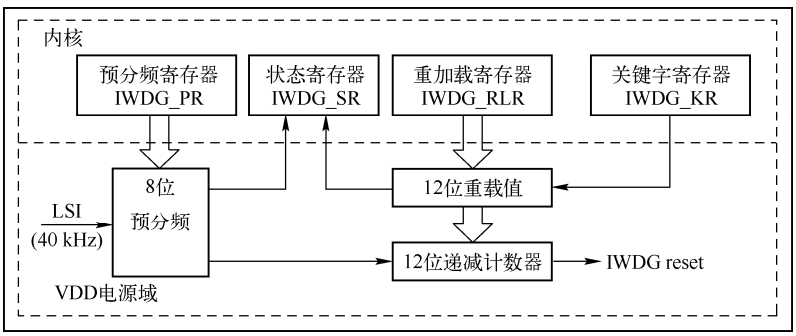


图 9-1　独立看门狗模块的功能框图

注：无论是在 Stop 模式下还是在 Standby 模式下，看门狗功能由内核供电区域供电。

独立看门狗由一个独立的 RC 振荡器驱动（在 Standby 和 Stop 状态下仍可操作）。当发生向下递减计数器的值达到 0 或向下递减计数器的值处于窗口值之外时，被重新加载时会产生复位。

当向独立看门狗的关键字寄存器写入启动指令 0x0000 CCCC 的时候，看门狗计数器开始由复位值 0xFFF 向下计数。当计数值达到 0x000 的时候，由独立看门狗发出复位信号。

任何时候将关键字 0x0000AAAA 写到 IWDG_KR 寄存器中，都会使得 IWDG_RLR 寄存器中的值被重加载到看门狗计数器中，从而阻止即将发生的复位动作。

1．窗口选项

IWDG 也能够工作在窗口看门狗模式下，只要在 IWDG_WINR 寄存器中设置适当的值即可。如果重加载操作执行的同时，看门狗计数器的值超出了窗口寄存器（IWDG_WINR）中存储的值，也会引起复位操作。IWDG_WINR 的默认值是 0x0000 0FFF，所以如果没有改写它，那么窗口选项默认是关闭的。窗口值一旦改变，立即就会引起看门狗计数器的一次重加载动作，将其置为 IWDG_RLR 中所设置的值，从而一定程度上延缓目前到下次复位所需的时间周期。

（1）当窗口选项使能时配置 IWDG

- 将 0x0000 CCCC 写到 IWDG_KR 寄存器，使能 IWDG。
- 向 IWDG_KR 寄存器写 0x0000 5555，打开寄存器访问许可。
- 向 IWDG_PR 写 0～7 的值，以配置 IWDG 的预分频器。
- 配置重加载寄存器（IWDG_RLR）。
- 等待状态寄存器 IWDG_SR 的值更新为 0x0000 0000。
- 配置窗口寄存器 IWDG_WINR。这将会引起自动将 IWDG_RLR 的值更新到看门狗计数器。

（2）当窗口选项被禁止时配置 IWDG

当窗口选项未被使用，可按下列顺序配置 IWDG。

- 向 IWDG_KR 寄存器写 0x0000 5555 打开寄存器访问许可。
- 向 IWDG_PR 写 0～7 的值，以配置 IWDG 的预分频器。
- 配置重加载寄存器（IWDG_RLR）。
- 等待状态寄存器 IWDG_SR 的值更新为 0x0000 0000。
- 将 IW_RLR 的值刷新到看门狗定时器（IWDG_KR = 0x0000 AAAA）。
- 将 0x0000 CCCC 写到 IWDG_KR 寄存器，使能 IWDG。

2．硬件看门狗

如果在 OPTION 字节中打开了“硬件看门狗”功能，那么在上电的时候看门狗就被自动打开。如果没有在看门狗计数器计数结束，或者向下计数的值超出窗口之前向关键字寄存器写入正确的值，会产生硬件复位请求。

3．寄存器访问保护

默认条件下，对 IWDG_PR、IWDG_RLR 和 IWDG_WINR 的写访问操作都是受保护的。想要改变这一点，必须先向 IWDG_KR 写入 0x0000 5555 解锁码。如果写入别的值，将会打破这个顺序，使得对寄存器的访问保护重新生效。这意味着在做重加载操作的时候（向该寄存器写入 0x0000 AAAA）就属于这种情况。可以通过一个状态寄存器观察预分频器的更新、看门狗计数器的重加载或窗口值的重加载。

4．调试模式

当微控制器进入调试模式时（内核被暂停），看门狗计数器可以继续运行，也可以被停止。这取决于 DBG 模块中的 DBG_IWDG_STOP 选项的配置。

9.3　窗口看门狗（WWDG）

窗口看门狗通常被用来监测由外部干扰或不可预见的逻辑条件造成的应用程序背离正常的运行序列而产生的软件故障。除非递减计数器的值在 T6 位变成 0 前被刷新，看门狗电路在达到预置的时间周期时，会产生一个 MCU 复位。在递减计数器达到窗口寄存器数值之前，如果 7 位的递减计数器数值（在控制寄存器中）被刷新，那么也将产生一个 MCU 复位。这表明递减计数器需要在一个有限的时间窗口中被刷新。

WWDG 时钟从 APB1 时钟预分频，并有一个可配置的时间窗口，这可通过编程来检测异常推迟或提前的应用行为。WWDG 最适合要求看门狗在一个精确时间窗口内做出反应的应用程序。

1．主要特性

（1）可编程的自由运行递减计数器。

（2）复位条件。

- 当递减计数器的值小于 0x40，（若看门狗被启动）则产生复位。
- 当递减计数器在窗口外被重新装载，（若看门狗被启动）则产生复位。

（3）提前唤醒中断（EWI）：如果启动了看门狗，当递减计数器等于 0x40 时产生提前唤醒中断。

2．功能描述（见图 9-2）

如果看门狗被启动（WWDG_CR 寄存器中的 WDGA 位被置“1”），并且当 7 位（T[6:0]）递减计数器从 0x40 翻转到 0x3F（T6 位清零）时，则产生一个复位。如果软件在计数器值大于窗口寄存器中的数值时重新装载计数器，将产生一个复位。

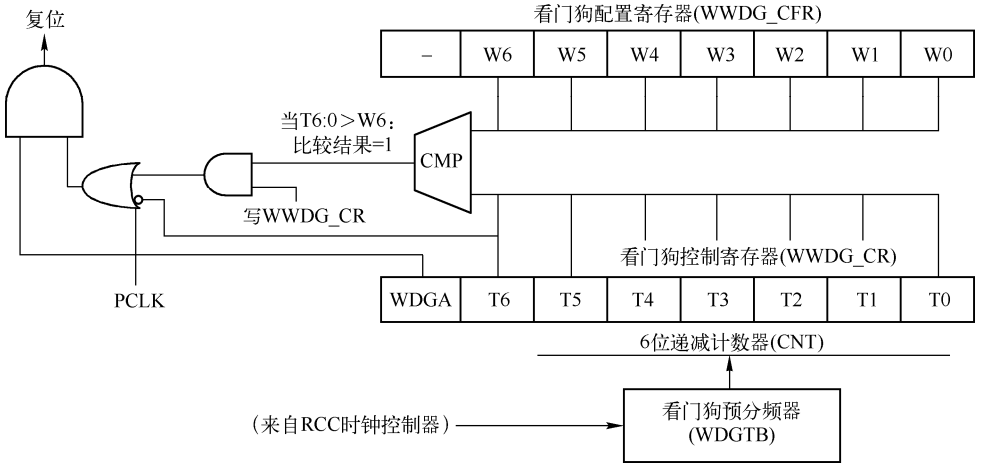


图 9-2　窗口看门狗框图

应用程序在正常运行过程中必须定期地写入 WWDG_CR 寄存器以防止 MCU 发生复位。只有当计数器值小于窗口寄存器的值时，才能进行写操作。储存在 WWDG_CR 寄存器中的数值必须在 0xFF～0xC0 之间。

在系统复位后，看门狗总是处于关闭状态，设置 WWDG_CR 寄存器的 WDGA 位能够开启看门狗，随后它不能再被关闭，除非发生复位。

递减计数器处于自由运行状态：即使看门狗被禁止，递减计数器仍继续递减计数。当看门狗被启用时，T6 位必须被设置，以防止立即产生一个软件复位。

T[5:0]位包含了看门狗产生复位之前的计时数目；复位前的延时时间在一个最小值和一个最大值之间变化，这是因为写入 WWDG_CR 寄存器时，预分频值是未知的。配置寄存器（WWDG_CFR）中包含窗口的上限值：要避免产生复位，递减计数器必须在其值小于窗口寄存器的数值并且大于 0x3F 时被重新装载，如果在实际复位产生之前必须进行特定的安全操作或数据记录，可以用提前唤醒中断。设置 WWDG_CFR 寄存器中的 WEI 位开启该中断。在复位之前当递减计数器到达 0x40 时，则产生此中断，同时可以用相应的中断服务程序（ISR）来触发特定的行为（例如通信或数据记录）。

在某些应用中，提前唤醒中断可以用来管理软件系统检测和/或系统恢复/故障弱化，但不产生 WWDG 复位。在这种情况下，相应的中断服务程序（ISR）将重加载 WWDG 计数器，以避免 WWDG 复位，然后触发必要的行为。

9.4 固 件 库

9.4.1 IWDG API

stm32f0xx_iwdg.c 文件提供了 IWDG 的库函数，该文件主要提供了预分频与计数器配置、IWDG 激活、标志管理。表 9-2 是 IWDG 的 API 函数。

表 9-2 IWDG 的 API 函数

函 数	描 述
IWDG_WriteAccessCmd	使能或屏蔽 IWDG_PR 和 IWDG_RLR 寄存器的写保护
IWDG_SetPrescaler	设置 IWDG 预分频值
IWDG_SetReload	设置 IWDG 预装载值
IWDG_ReloadCounter	重加载 IWDG 计数值
IWDG_SetWindowValue	设置 IWDG 窗口值
IWDG_Enable	使能 IWDG
IWDG_GetFlagStatus	检验 IWDG 标志是否被设置

根据窗口选项是否使能，使用 IWDG 外设分为两种情况。

（1）窗口选项使能

- 当在软件模式（不需使能 LSI，被硬件使能）使用 IWDG，使用 IWDG_Enable()启用 IWDG。
- 使用 IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable)函数使能 IWDG_PR 和 IWDG_RLR 寄存器的写保护。
- 使用 IWDG_SetPrescaler()函数配置 IWDG 预分频。

- 使用 IWDG_SetReload()函数配置 IWDG 计数器值。
- 使用 IWDG_GetFlagStatus()函数等待 IWDG 寄存器被更新。
- 使用 IWDG_SetWindowValue()函数配置看门狗窗口值。

（2）窗口选项被禁用

- 当 IWDG 用于软件模式（不需使能 LSI，被硬件使能），使用 IWDG_Enable()使能 IWDG。
- 使用 IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable)函数使能 IWDG_PR 和 IWDG_RLR 寄存器的写保护。
- 使用 IWDG_SetPrescaler()函数配置 IWDG 预分频。
- 使用 IWDG_SetReload()函数配置 IWDG 计数器值。
- 使用 IWDG_GetFlagStatus()函数等待 IWDG 寄存器被更新。
- 使用 IWDG_ReloadCounter()函数重新加载 IWDG 计数器阻止控制器复位。
- 当在软件模式（不需使能 LSI，被硬件使能）使用 IWDG，使用 IWDG_Enable()启用 IWDG。

9.4.2　WWDG 固件库

WWDG 的固件库函数位于 stm32f0xx_wwdg.c 中，提供了预分频、刷新窗口和计数器配置、WWDG 激活、中断和标志管理。表 9-3 是 WWDG 的 API 函数。

表 9-3　WWDG API

函　　数	描　　述
WWDG_DeInit	重初始化 WWDG 外设寄存器为默认复位值
WWDG_SetPrescaler	设置 WWDG 预分频
WWDG_SetWindowValue	设置 WWDG 窗口值
WWDG_EnableIT	使能 WWDG 提前唤醒中断
WWDG_SetCounter	设置 WWDG 计数器值
WWDG_Enable	使能 WWDG 和加载计数器值
WWDG_GetFlagStatus	检查提前唤醒中断标志置位与否
WWDG_ClearFlag	清除提前唤醒中断标志

使用 WWDG 的驱动程序过程如下：

① 使用 RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE)函数使能 WWDG 时钟。

② 使用 WWDG_SetPrescaler()函数配置 WWDG 预分频。

③ 使用 WWDG_SetWindowValue()函数配置 WWDG 窗值。

④ 设置 WWDG 计数器值，使用 WWDG_Enable()启动。当 WWDG 使能时，计数器值应配置一个大于 0x40 的值，防止立即复位情况。

⑤ 当计数器达到 0x40 可使能提前唤醒中断。

⑥ 在程序中以固定间隔使用 WWDG_SetCounter()刷新 WWDG 计数器阻止复位。

9.5 看门狗实例

下面程序使用 WWDG，喂狗时间间隔需要小于 43.7ms。在主循环中，间隔 30ms 喂狗一次。喂狗 10 次后，模拟 HardFault，产生系统复位。其中 HardFault 是通过堆栈溢出方式模拟的。

```
//模拟 HardFault
void StackFlow(void)
{
  int a[10],i;
     for(i=0; i<10000; i++)
  {       a[i]=1; }

}
int main(void)
{
     int WatchCounter =0;
          USART_Configuration();
     printf ("------------Programm Beinging---------- \n\r") ;
   /*  判断是否来自 WWDG 复位*/
   if (RCC_GetFlagStatus(RCC_FLAG_WWDGRST) != RESET)
               {
                printf("WWDG Reset\n\r") ;
                       /*  清除复位标志  */
                RCC_ClearFlag();

   }
   else
   {
      printf("Go to WWWDG\n\r");
         /*  配置 WWDG */
              WWDG_Config();

    }
   /*    设置 SysTick 定时器：1ms 间隔    */
   if (SysTick_Config(SystemCoreClock / 1000))
   {
      while (1)
      {}
   }
   while (1)
   { /*  更新 WWDG 计数器  */
                        WWDG_SetCounter(127);
              /*  在连续一段喂狗后，触发 HardFault */
```

```
                    if   (WatchCounter++<10)
                    {
                    Delay(33);
                    }
                    else
                    {
                            StackFlow();

                    }
            }
}

/*配置 WWDG */
static void WWDG_Config(void)
{
  /*  使能 WWDG */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);
  /* WWDG 时钟计数  = (PCLK1 (48MHz)/4096)/8 = 1464Hz (～683 us)    */
  WWDG_SetPrescaler(WWDG_Prescaler_8);
  /*设置窗口值为 80；只有当计数器值低于 80（并大于 64）WWDG 计数器更新，否则复位  */
  WWDG_SetWindowValue(80);
  /*  使能  WWDG，设置计数器值 127，WWDG 的超时=683×64=43.7ms
    683 * (127-80)= 32.1ms<刷新窗<683 * 64 = 43.7ms           */
  WWDG_Enable(127);
}
```

9.6　小　　结

STM 的产品存在两个看门狗，用法上差异不是很大。二者最主要的差异是时钟源，独立看门狗的时钟源是 LSI，从而使得它适合主时钟故障，但要求看门狗仍旧可工作的场合。CPU 内部是多个模块共同工作的，它不是完全意义上的独立整体，就会存在部分功能失效情况。在一些强干扰情况，会容易出现 CPU 部分“死”（假设一个产品包含显示屏显示、ADC 采样中断功能，出现了屏幕不更新数据，但 ADC 中断正常的情况）。偏偏出现屋漏逢雨天情况：喂狗程序碰巧放在中断中。这种情况 CPU 既不能正常工作，又无法复位。如果碰巧 APB1 的时钟所关联部分出问题了，即窗口看门狗不进行计数了，窗口看门狗就无法进行复位。对于这种情况，独立看门狗就可以很好地发挥作用，通过合理布局，来解决主时钟失效情况。但不管怎么说，通过看门狗解决 CPU“死”的问题，属于最后一道防线。

第10章

定 时 器

STM32F0 系列控制器拥有 6 个通用定时器（STM32F030x 是 5 个通用定时器）、1 个基本定时器（STM32F072xx 是 2 个基本定时器）和 1 个高级控制定时器。为了更好地理解定时器的用法，可将定时器想象成一块手表，即一个基础时钟信号（时基）一直在默默无闻运行着。

首先计划多少个时基后执行某一个动作（即中断），设定后好，开始计数。这种方式就类似闹钟，是通常所说定时器功能（例如 TIM6/TIM7，只有基本定时功能）。

将 PWM 理解成两个闹钟在运行，一长一短，占空比是短闹钟与长闹钟的比值。如果设定的时基到了，通过 CPU 的引脚输出一个高/低电平，则为 PWM 输出模式，当然可能存在一次性的（即单次 PWM）。在输出 PWM 的过程中，偶尔需要来个紧停，即刹车功能。PWM 功能应用场合主要集中在电力电子以及电机控制领域。

PWM 是输出定时信号，属于闹钟的扩展用法。对应的就有一个秒表功能，即输入捕获功能，用于对两次输入的信号计时。

时基信号来源是多样化的（STM32F0 的时基信号可以是内部时钟、外部时钟以及其他定时器的输出），而对时基信号的计数方式也可以多样化（STM32F0 的计数方式有向上计数、向下计数以及中央对齐方式）。

10.1 STM32F0 定时器实现

STM32F0 系列的定时器相互对比见表 10-1。

表 10-1 STM32F0x 定时器实现

定时器类型	定时器	计数器分辨率	计数器类型	预分频因子	DMA 请求生成	捕获/比较通道	互补输出
高级控制	TIM1	16 位	递增； 递减； 递增/递减	1～65536 之间的整数	是	4	是
通用类型	TIM2	32 位	递增； 递减； 递增/递减	1～65536 之间的整数	是	4	否
	TIM3	16 位	递增； 递减； 递增/递减	1～65536 之间的整数	是	4	否

续表

定时器类型	定时器	计数器分辨率	计数器类型	预分频因子	DMA 请求生成	捕获/比较通道	互补输出
	TIM14	16 位	递增	1～65536 之间的整数	否	1	否
	TIM15	16 位	递增	1～65536 之间的整数	是	2	是
	TIM16, TIM17	16 位	递增	1～65536 之间的整数	是	1	是
基本类型	TIM6	16 位	递增	1～65536 之间的整数	是	0	否

因高级定时器 TIM1 涵盖其他几种定时器的功能，所以本章是以高级定时器为基础进行讲解的。高级控制定时器（TIM1）可看作基于 6 通道的三相 PWM，带有可编程死区的互补 PWM 输出。4 个独立通道可用来输入捕获、输出比较器、PWM 生成（沿或者中间对齐模式）以及单脉冲输出。

如果配置成 16 位 PWM 发生器，占空比可以是 0%～100%。调试模式下计数器可被冻结，拥有标准定时器的特征。通过定时器连接特性，高级定时器与其他定时器协作实现同步和事件链。

通用定时器（TIM3、TIM14～TIM17）可用于生成 PWM 或者单脉冲模式输出、通道捕获/输出比较，以及简单的定时器功能。其中 TIM3 定时器可用于处理正交编码以及霍尔传感器。

基本类型定时器用于通用的 16 位定时功能，其中 STM32F07x 系列的基本定时器可用于 DAC 触发器发生。

10.2　功 能 描 述

图 10-1 是 TIM1 的原理框图。TIM1 定时器的功能如下。

- 16 位向上、向下、向上/下自动装载计数器。
- 16 位可编程（可以实时修改）预分频器，计数器时钟频率的分频系数为 1～65535 之间的任意数值。
- 多达 4 个独立通道：输入捕获、输出比较、PWM 生成（边缘或中间对齐模式）、单脉冲模式输出。
- 死区时间可编程的互补输出。
- 使用外部信号控制定时器和定时器互联的同步电路。
- 允许在指定数目的计数器周期之后更新定时器寄存器的重复计数器。
- 刹车输入信号可以将定时器输出信号置于复位状态或者一个已知状态。
- 如下事件发生时产生中断/DMA。
  - 更新：计数器向上溢出/向下溢出，计数器初始化（通过软件或者内部/外部触发）。
  - 触发事件（计数器启动、停止、初始化或者由内部/外部触发计数）。
  - 输入捕获。
  - 输出比较。
  - 刹车信号输入。
- 支持用于定位的增量（正交）编码器和霍尔传感器电路。
- 触发输入作为外部时钟或者按周期的电流管理。

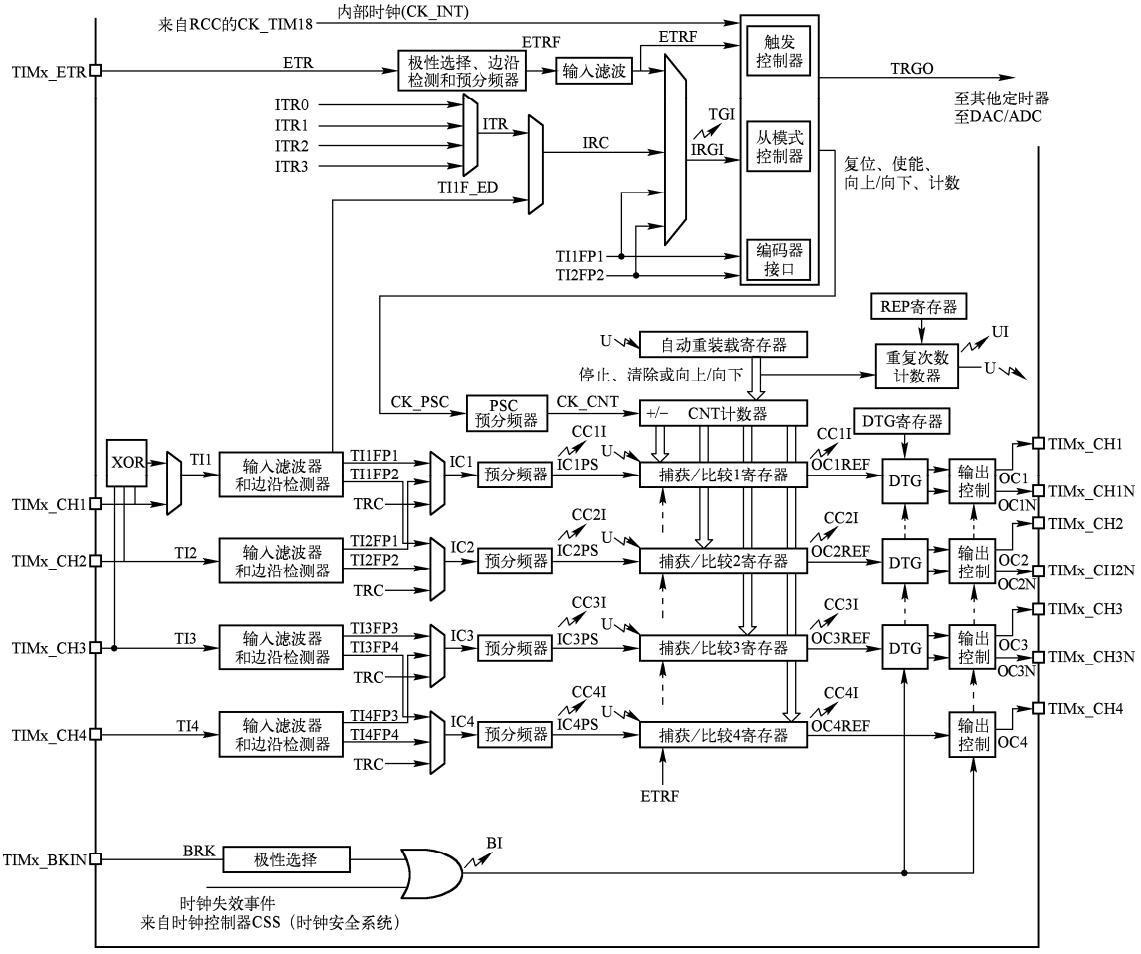


图 10-1　高级控制定时器框图

注：根据控制位的设定，在 U（更新）事件时传送预加载寄存器的内容至工作寄存器。

：事件。

：中断和 DMA 输出。

10.2.1　时基单元

可编程高级控制定时器的主要部分是一个 16 位计数器和与其相关的自动装载寄存器。这个计数器可以向上计数、向下计数或者向上/向下双向计数。此计数器时钟由预分频器分频得到。计数器、自动装载寄存器和预分频器寄存器可以由软件读/写，即使计数器还在运行读/写仍然有效。时基单元包括计数器寄存器（TIMx_CNT）、预分频器寄存器（TIMx_PSC）、自动装载寄存器（TIMx_ARR）、重复次数寄存器（TIMx_RCR）。

自动装载寄存器是预先装载的，写或读自动重装载寄存器将访问预装载寄存器。根据在 TIMx_CR1 寄存器中的自动装载预装载使能位（ARPE）的设置，预装载寄存器的内容被立即或在每次的更新事件 UEV 时传送到影子寄存器。当计数器达到溢出条件（或向下计数时的下溢条件），并当 TIMx_CR1 寄存器中的 UDIS 位等于 0 时，产生更新事件。更新事件也

可以由软件产生。

计数器由预分频器的时钟输出 CK_CNT 驱动，仅当设置了计数器 TIMx_CR1 寄存器中的计数器使能位（CEN）时，CK_CNT 才有效。

注意：在设置了 TIMx_CR 寄存器的 CEN 位的 1 个时钟周期后，计数器开始计数。

预分频器可以将计数器的时钟频率按 1～65 536 之间的任意值分频。它是基于一个（在 TIMx_PSC 寄存器中的）16 位寄存器控制的 16 位计数器。因为这个控制寄存器带有缓冲器，能够在运行时被改变。新的预分频器的参数在下一次更新事件到来时被采用。

10.2.2　计数器

1．向上计数模式

在向上计数模式中，计数器从 0 计数到自动加载值（TIMx_ARR 计数器的内容），然后重新从 0 开始计数并且产生一个计数器溢出事件。如果使用了重复计数器功能，在向上计数达到设置的重复计数次数（TIMx_RCR）时，才产生更新事件（UEV）；否则每次计数器溢出时都会产生更新事件。

在 TIMx_EGR 寄存器中（通过软件方式或者使用从模式控制器）设置 UG 位也同样可以产生一个更新事件。

通过软件设置 TIMx_CR1 寄存器中的 UDIS 位，可以禁止更新事件，这样可以避免在向预装载寄存器中写入新值时更新影子寄存器。在 UDIS 位被清 0 之前，将不产生更新事件。但是在应该产生更新事件时，计数器会被清 0，同时预分频器的计数也被清 0（但预分频器的数值不变）。此外，如果设置了 TIMx_CR1 寄存器中的 URS 位（选择更新请求），设置 UG 位将产生一个更新事件 UEV，但硬件不设置 UIF 标志（即不产生中断或 DMA 请求）。这是为了避免在捕获模式下清除计数器时，同时产生更新和捕获中断。当发生一个更新事件时，所有的寄存器都被更新，同时（依据 URS 位）设置更新标志位（TIMx_SR 寄存器中的 UIF 位）。

- 重复计数器被重新加载为 TIMx_RCR 寄存器的内容。
- 自动装载影子寄存器被重置预装载寄存器的值（TIMx_ARR）。
- 预分频器的缓冲区被写入预装载寄存器的值（TIMx_PSC 寄存器的内容）。

图 10-2 是 TIMx_ARR=0x36 时计数器在不同时钟频率下计数器的动作。

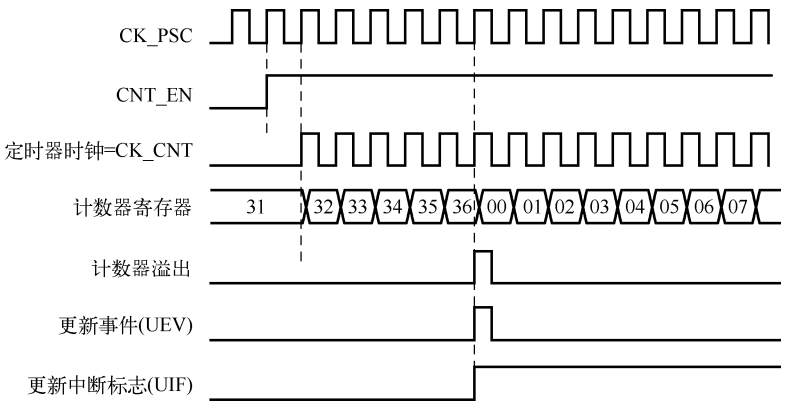


图 10-2　计数器时序图（内部时钟分频因子为 1）

2．向下计数模式

在向下模式中，计数器从自动装入的值（TIMx_ARR 计数器的值）开始向下计数到 0，然后从自动装入的值重新开始并且产生一个计数器向下溢出事件。如果使用了重复计数器，当向下计数重复了重复计数寄存器（TIMx_RCR）中设定的次数后，才产生更新事件（UEV），否则每次计数器下溢时都会产生更新事件。设置 TIMx_EGR 寄存器的（通过软件方式或者使用从模式控制器）UG 位，也同样可以产生一个更新事件。

TIMx_CR1 寄存器的设置影响更新事件，这部分与向上计数方式相同。

当发生一个更新事件时，所有的寄存器都被更新，同时（依据 URS 位）设置更新标志位（TIMx_SR 寄存器中的 IF 位）。

- 重复计数器被重新加载为 TIMx_RCR 寄存器的内容。
- 预分频器的缓冲区被写入预装载寄存器的值（TIMx_PSC 寄存器的内容）。
- 自动装载影子寄存器被重新为预装载寄存器的值（TIMx_ARR）。如果在计数器重载之前自动重装被更新，此时在下一个周期时才是预期的值。

3．重复计数器

更新事件（UEV）只能在重复计数达到 0 的时候产生。这个特性对产生 PWM 信号非常有用。这意味着在每 *N* 次计数上溢或下溢时，数据从预装载寄存器传输到影子寄存器（TIMx_ARR 自动重载入寄存器，TIMx_PSC 预装载寄存器，还有在比较模式下的捕获/比较寄存器 TIMx_CCRx），*N* 是 TIMx_RCR 重复计数寄存器中的值。

重复计数器在下述任一条件成立时递减：

- 向上计数模式下每次计数器溢出时。
- 向下计数模式下每次计数器下溢时。
- 中央对齐模式下每次上溢和每次下溢时。虽然这样限制了 PWM 的最大循环周期为 128，但它能够在每个 PWM 周期两次更新占空比。在中央对齐模式下，因为波形是对称的，如果每个 PWM 周期中仅刷新一次比较寄存器，则最大的分辨率为 2×Tck。

重复计数器是自动加载的，重复速率由 TIMx_RCR 寄存器的值定义。当更新事件由软件产生（通过设置 TIMx_EGR 中的 UG 位）或者通过硬件的从模式控制器产生，则无论重复计数器的值是多少，立即发生更新事件，并且 TIMx_RCR 寄存器中的内容被重载入重复计数器。

在中央对齐模式下，如果 RCR 是奇数，一旦 RCR 寄存器被写入并且计数器已经启动，每次溢出和下溢时都发生更新事件。如果 RCR 在计数器开始之前被写入，在溢出时发生 UEV 事件。如果 RCR 在计数器开始之后被写入，在下溢时发生 UEV 事件。例如 RCR =3，则在 RCR 被写入后的每 4 个溢出和下溢产生 UEV 事件。

10.2.3 时钟源

内部时钟（CK_INT）、外部时钟模式 1（外部输入引脚）、外部时钟模式 2（外部触发输入 ETR）、内部触发输入（ITRx，即使用一个定时器作为另一个定时器的预分频器）可以作为计数器时钟。

如果禁止了从模式控制器（SMS=000），则 CEN、DIR（TIMx_CR1 寄存器）和 UG 位（TIMx_EGR 寄存器）是事实上的控制位，并且只能被软件修改（除非 UG 位自动被清除）。

一旦 CEN 位被写成 1，预分频器的时钟就由内部时钟 CK_INT 提供。

图 10-3 显示了在不带预分频器时，控制电路和向上计数器在正常模式下的动作。

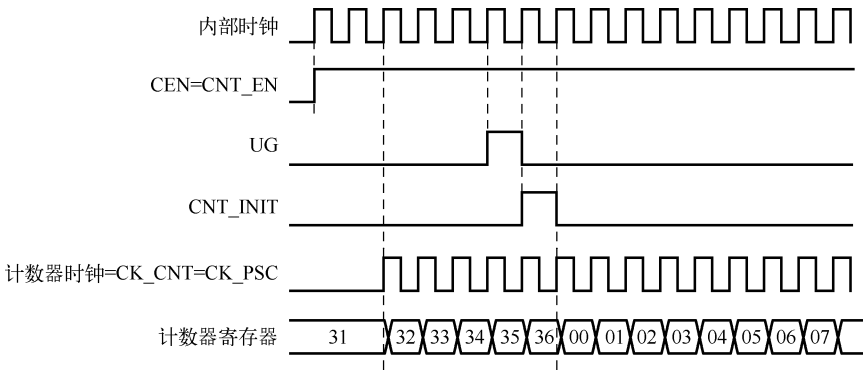


图 10-3　内部时钟分频是 1 时，正常模式下的控制电路

10.2.4　捕获/比较通道

每一个捕获/比较通道都是包括一个捕获/比较寄存器（包含影子寄存器）、一个捕获的输入部分（数字滤波、多路复用和预分频器）和一个输出部分（比较器和输出控制）。

图 10-4 是捕获/比较通道的输入部分。输入部分采样相对相应的 TIx 输入信号，产生一个滤波后的信号 TIxF。然后，一个带极性选择的边缘监测器产生一个信号（TIxFPx），它可以作为从模式控制器的输入触发或者作为捕获控制。该信号被预分频后进入捕获寄存器（ICxPS）。

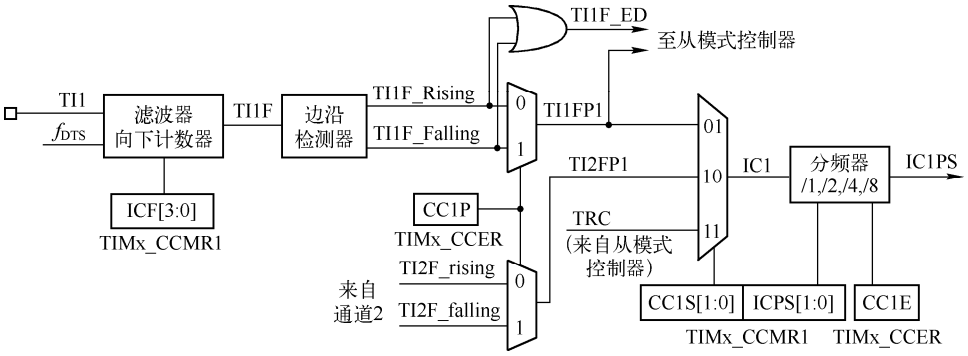


图 10-4　捕获/比较通道（如：通道 1 输入部分）

输出部分产生一个中间波形 OCxRef（高有效）作为基准，链的末端决定最终输出信号的极性，如图 10-5 所示。

捕获/比较模块由一个预装载寄存器和一个影子寄存器组成。读/写过程仅操作预装载寄存器。在捕获模式下，捕获发生在影子寄存器上，然后再复制到预装载寄存器中。在比较模式下，预装载寄存器的内容被复制到影子寄存器中，然后影子寄存器的内容和计数器进行比较。

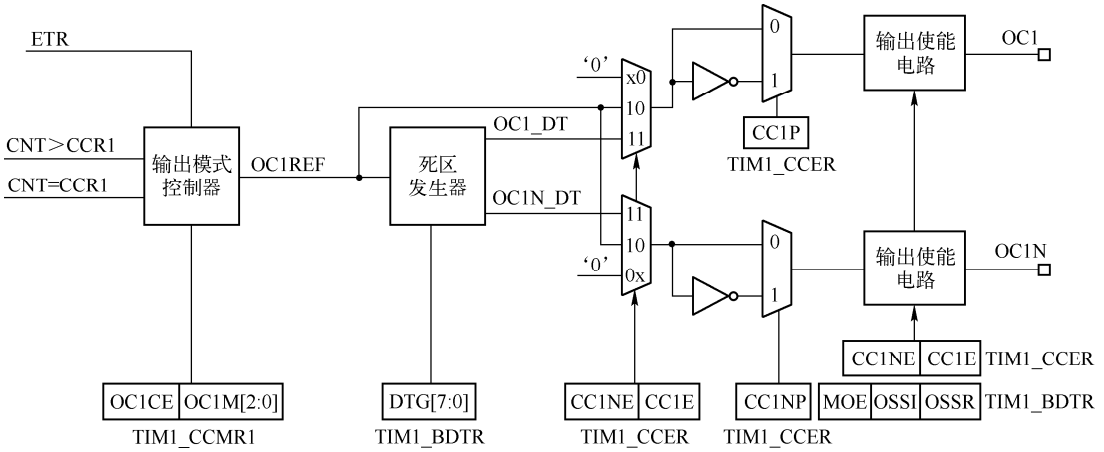


图 10-5　捕获/比较通道的输出部分（通道 1～3）

10.2.5　输入捕获模式

在输入捕获模式下，当检测到 ICx 信号上相应的边沿跳变时，计数器的当前值被锁存到捕获/比较寄存器（TIMx_CCRx）中。当发生捕获事件时，相应的 CCxIF 标志（TIMx_SR 寄存器）被置 1。如果开放了中断或者 DMA，则产生一个中断或者一个 DMA 请求。如果发生捕获事件时 CCxIF 标志已经为高，那么重复捕获标志 CCxOF（TIMx_SR 寄存器）被置 1。软件写 CCxIF=0 可清除 CCxIF，或读取存储在 TIMx_CCRx 寄存器中的捕获数据也可清除 CCxIF。写 CCxOF=0 可清除 CCxOF。

以下例子说明如何在 TI1 输入的上升沿时捕获计数器的值到 TIMx_CCR1 寄存器中，步骤如下。

① 选择有效输入端：TIMx_CCR1 必须链接到 TI1 输入，所以写入 TIMx_CCR1 寄存器中的 CC1S=01，只要 CC1S 不为 00，通道被配置为输入并且 TIMx_CCR1 寄存器变为只读。

② 根据连接到计数器的输入信号特点，配置输入滤波器为所需的带宽（输入为 TIx 时，输入滤波器控制位是 TIMx_CCMRx 寄存器中的 ICxF 位）。假设输入信号在最多 5 个内部时钟周期的时间内抖动，我们要配置滤波器的带宽长于 5 个时钟周期。因此，我们可以（以 f_{DTS} 频率采样）连续采样 8 次，以确认在 TI1 上一次新的边沿变换，然后在 TIMx_CCMR1 寄存器中写入 IC1F=0011。

③ 选择 TI1 通道的有效转换边沿。通过向 TIMx_CCER 寄存器中写入 CC1P=0 和 CC1NP=0（本例中为上升沿）。

④ 配置输入预分频器。在本例中，我们希望在每个有效的电平转换时发生一次捕获，因此预分频器被禁止（写 TIMx_CCMR1 寄存器的 IC1PS=00）。

⑤ 写 TIMx_CCER 寄存器的 CC1E=1，允许捕获计数器的值到捕获寄存器中。

⑥ 如果需要，通过设置 TIMx_DIER 寄存器中的 CC1IE 位允许相关中断请求，通过设置 TIMx_DIER 寄存器中的 CC1DE 位允许 DMA 请求。

当发生一个输入捕获时，发生如下过程。

① 产生有效的电平转换时，计数器的值被传送到 TIMx_CCR1 寄存器。

② CC1IF 标志被设置（中断标志）。当发生至少 2 个连续的捕获时，而 CC1IF 未曾被清除，CC1OF 也被置 1。

③ 如设置了 CC1IE 位，则会产生一个中断。

④ 如设置了 CC1DE 位，则还会产生一个 DMA 请求。

为了处理捕获溢出，建议在读出捕获溢出标志之前读取数据，这是为了避免丢失在读出捕获溢出标志之后和读取数据之前可能产生的捕获溢出。

PWM 输入模式是输入捕获模式的一个特例。

10.2.6　强制输出模式

在输出模式（TIMx_CCMRx 寄存器中 CCxS=00）下，输出比较信号（OCxREF 和相应的 OCx/OCxN）能够直接由软件强制为有效或无效状态，而不依赖于输出比较寄存器和计数器间的比较结果。

置 TIMx_CCMRx 寄存器中相应的 OCxM=101，即可强制输出比较信号（OCxREF/OCx）为有效状态。这样 OCxREF 被强制为高电平（OCxREF 始终为高电平有效），同时 OCx 得到 CCxP 极性相反的信号。例如：CCxP=0（OCx 高电平有效），则 OCx 被强制为高电平。置 TIMx_CCMRx 寄存器中的 OCxM=100，可强制 OCxREF 信号为低。该模式下，在 TIMx_CCRx 影子寄存器和计数器之间的比较仍然在进行，相应的标志也会被修改。因此仍然会产生相应的中断和 DMA 请求。

10.2.7　输出比较模式

此项功能是用来控制一个输出波形，或者指示一段给定的时间已经到时。当计数器与捕获/比较寄存器的内容相同时，输出比较功能做如下操作。

① 将输出比较模式（TIMx_CCMRx 寄存器中的 OCxM 位）和输出极性（TIMx_CCER 寄存器中的 CCxP 位）定义的值输出到对应的引脚上。在比较匹配时，输出引脚可以保持它的电平（OCxM=000）、被设置成有效电平（OCxM=001）、被设置成无效电平（OCxM=010）或进行翻转（OCxM=011）。

② 设置中断状态寄存器中的标志位（TIMx_SR 寄存器中的 CxIF 位）。

③ 若设置了相应的中断屏蔽（TIMx_DIER 寄存器中的 CCxIE 位），则产生一个中断。

④ 若设置了相应的使能位（TIMx_DIER 寄存器中的 CCxDE 位，TIMx_CR2 寄存器中的 CCDS 位选择 DMA 请求功能），则产生一个 DMA 请求。

TIMx_CCMRx 中的 OCxPE 位选择 TIMx_CCRx 寄存器是否需要使用预装载寄存器。在输出比较模式下，更新事件 UEV 对 OCxREF 和 OCx 输出没有影响。同步的精度可以达到计数器的一个计数周期。输出比较模式（在单脉冲模式下）也能用来输出一个单脉冲。输出比较模式的配置步骤如下。

① 选择计数器时钟（内部，外部，预分频器）。

② 将相应的数据写入 TIMx_ARR 和 TIMx_CCRx 寄存器中。

③ 如果要产生一个中断请求，设置 CCxIE 位。

④ 选择输出模式，例如：

- 要求计数器与 CCRx 匹配时翻转 OCx 的输出引脚，设置 OCxM=011。

- 置 OCxPE = 0 禁用预装载寄存器。
- 置 CCxP = 0 选择极性为高电平有效。
- 置 CCxE = 1 使能输出。

⑤ 设置 TIMx_CR1 寄存器的 CEN 位启动计数器。

TIMx_CCRx 寄存器能够在任何时候通过软件进行更新以控制输出波形，条件是未使用预装载寄存器（OCxPE=0，否则 TIMx_CCRx 的影子寄存器只能在发生下一次更新事件时被更新）。图 10-6 给出了一个例子。

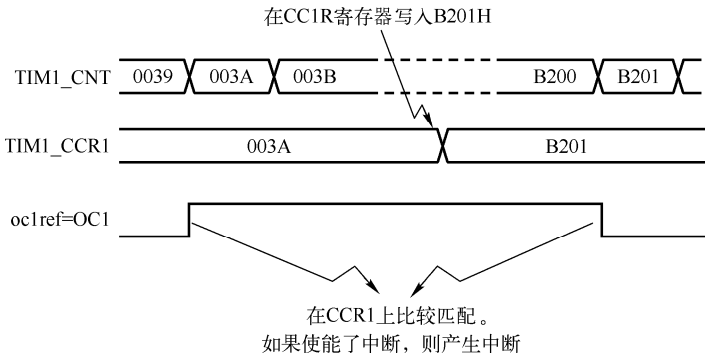


图 10-6　输出比较模式，翻转 OC1

10.2.8　PWM 模式

脉冲宽度调制模式可以产生一个由 TIMx_ARR 寄存器确定频率和由 TIMx_CCRx 寄存器确定占空比的信号。在 TIMx_CCMRx 寄存器中的 OCxM 位写入 110（PWM 模式 1）或 111（PWM 模 2），能够独立地设置每个 OCx 输出通道产生一路 PWM。必须通过设置 TIMx_CCMRx 寄存器的 OCxPE 位使能相应的预装载寄存器，最后还要设置 TIMx_CR1 寄存器的 ARPE 位（在向上计数或中心对称模式中）使能自动重装载的预装载寄存器。仅当发生一个更新事件的时候，预装载寄存器才能被传送到影子寄存器，因此在计数器开始计数之前，必须通过设置 TIMx_EGR 寄存器中的 UG 位来初始化所有的寄存器。OCx 的极性可以通过软件在 TIMx_CCER 寄存器中的 CCxP 位设置，它可以设置为高电平有效或低电平有效。OCx 的输出使能通过（TIMx_CCER 和 TIMx_BDTR 寄存器中）CCxE、CCxNE、MOE、OSSI 和 OSSR 位的组合控制。

在 PWM 模式（模式 1 模式 2）下，TIMx_CNT 和 TIMx_CCRx 始终在进行比较（依据计数器的计数方向）以确定是否符合 TIMx_CCRx≤TIMx_CNT 或者 TIMx_CNT≤TIMx_CCRx。根据 TIMx_CR1 寄存器中 CMS 位的状态，定时器能够产生边沿对齐的 PWM 信号或中央对齐的 PWM 信号。

1．PWM 边沿对齐模式

当 TIMx_CR1 寄存器中的 DIR 位为低的时候执行向上计数。当 TIMx_CNT<TIMx_CCRx 时，PWM 参考信号 OCxREF 为高，否则为低。如果 TIMx_CCR 中的比较值大于自动重装载值（TIMx_ARR），则 OCxREF 保持为 1。如果比较值为 0，则 OCxREF 保持为 0。图 10-7 为 TIMx_ARR=8 时边沿对齐的 PWM 波形实例。

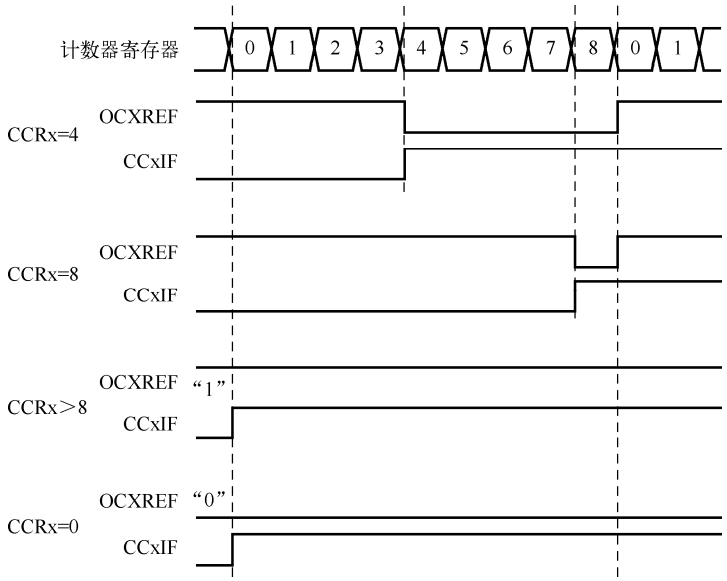


图 10-7　边沿对齐的 PWM 波形（ARR=8）

当 TIMx_CR1 寄存器的 DIR 位为高时执行向下计数。在 PWM 模式 1，当 TIMx_CNT> TIMx_CCRx 时参考信号 OCxREF 为低，否则为高。如果 TIMx_CCRx 中的比较值大于 TIMx_ARR 中的自动重装载值，则 OCxREF 保持为 1。该模式下不能产生 0%的 PWM 波形。

2. PWM 中央对齐模式

当 TIMx_CR1 寄存器中的 CMS 位不为 00 时，为中央对齐模式（所有其他的配置对 OCxREF/OCx 信号都有相同的作用）。根据不同的 CMS 位设置，比较标志可以在计数器向上计数时，在计数器向下计数时，或在计数器向上和向下计数时被置 1。TIMx_CR1 寄存器中的计数方向位（DIR）由硬件更新，不要用软件修改它。图 10-8 给出了一些中央对齐的 PWM 波形的例子，其中 TIMx_ARR=8，PWM 模式为 1，TIMx_CR1 寄存器的 CMS=01，在中央对齐模式 1 下，当计数器向下计数时设置比较标志。

使用中央对齐模式的提示：

- 进入中央对齐模式时，使用当前的向上/向下计数配置，这就意味着计数器向上还是向下计数取决于 TIMx_CR1 寄存器中 DIR 位的当前值。此外，软件不能同时修改 DIR 和 CMS 位。
- 不推荐当运行在中央对齐模式时改写计数器，因为这会产生不可预知的结果。特别是如下情况下：
  - 如果写入计数器的值大于自动重加载的值（TIMx_CNT>TIMx_ARR），则方向不会被更新。例如，如果计数器正在向上计数，它就会继续向上计数。
  - 如果将 0 或者 TIMx_ARR 的值写入计数器，方向被更新，但不产生更新事件 UEV。
- 使用中央对齐模式最保险的方法，就是在启动计数器之前产生一个软件更新（设置 TIMx_EGR 位中的 UG 位），并且不要在计数进行过程中修改计数器的值。

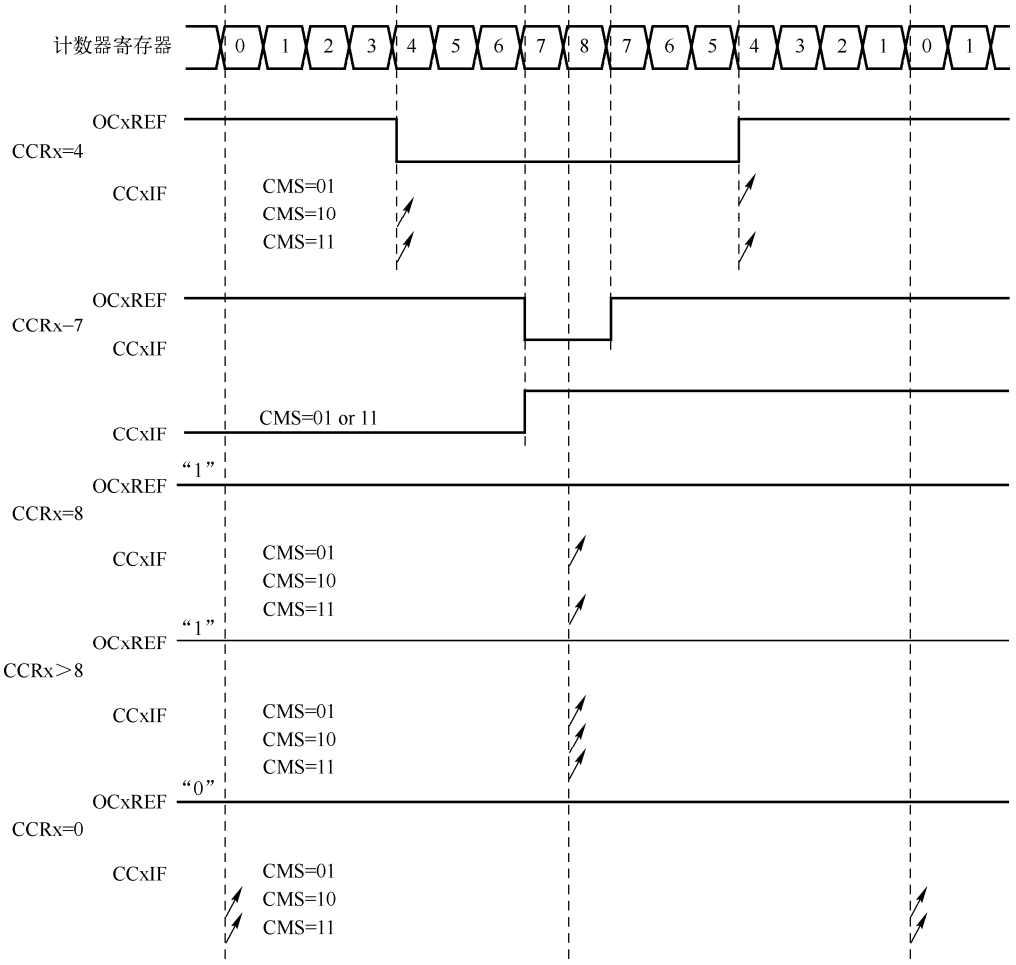


图 10-8　中央对齐的 PWM 波形（APR=8）

10.2.9　互补输出和死区插入

高级控制定时器（TIM1）能够输出两路互补信号，并且能够管理输出的瞬时关断和接通。这段时间通常被称为死区，用户应该根据连接的输出器件和它们的特性（电平转换的延时、电源开关的延时等）来调整死区时间。配置 TIMx_CCER 寄存器中的 CCxP 和 CCxNP 位，可以为每一个输出独立地选择极性（主输出 OCx 或互补输出 OCxN）。互补信号 OCx 和 OCxN 通过下列控制位的组合进行控制：TIMx_CCER 寄存器的 CCxE 和 CCxNE 位，TIMx_BDTR 和 TIMx_CR2 寄存器中的 MOE、OISx、OISxN、OSSI 和 OSSR 位，特别的是，在转换到 IDLE 状态时（MOE 下降到 0）死区被激活。同时设置 CCxE 和 CCxNE 位将插入死区，如果存在刹车电路，则还要设置 MOE 位。每一个通道都有一个 10 位的死区发生器。参考信号 OCxREF 可以产生 2 路输出 OCx 和 OCxN。如果 OCx 和 OCxN 为高有效：

- OCx 输出信号与参考信号相同，只是它的上升沿相对于参考信号的上升沿有一个延迟。

- OCxN 输出信号与参考信号相反，只是它的上升沿相对于参考信号的下降沿有一个延迟。

如果延迟大于当前有效的输出宽度（OCx 或者 OCxN），则不会产生相应的脉冲。

图 10-9～图 10-11 显示了死区发生器的输出信号和当前参考信号 OCxREF 之间的关系（假设 CCxP=0、CCxNP=0、MOE=1、CCxE=1，并且 CCxNE=1）。

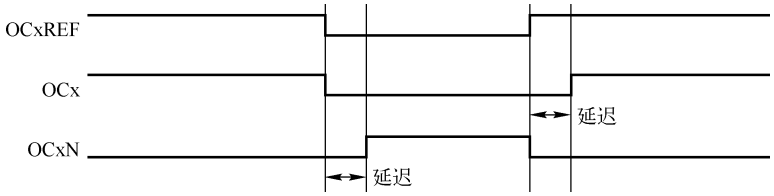


图 10-9　带死区插入的互补输出

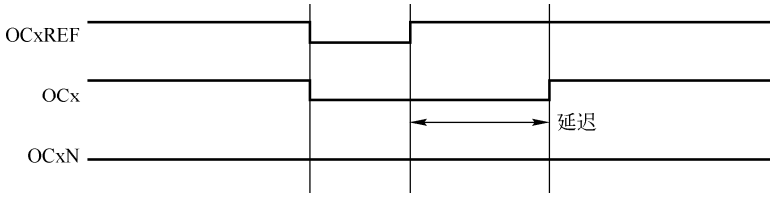


图 10-10　死区波形延迟大于负脉冲

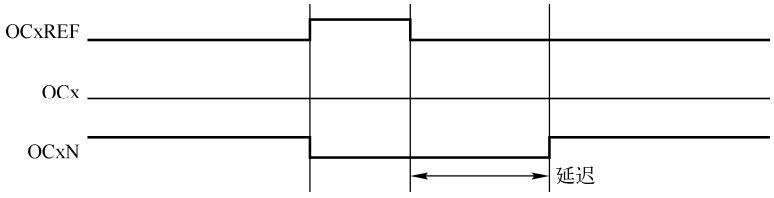


图 10-11　死区波形延迟大于正脉冲

每一个通道的死区延时都是相同的，由 TIMx_BDTR 寄存器中的 DTG 位编程配置。

在输出模式下（强制、输出比较或 PWM），通过配置 TIMx_CCER 寄存器的 CCxE 和 CCxNE 位，OCxREF 可以被重定向到 OCx 或者 OCxN 的输出。

这个功能可以在互补输出处于无效电平时，在某个输出上送出一个特殊的波形（例如 PWM 或者静态有效电平）。另一个作用是，让两个输出同时处于无效电平，或处于有效电平和带死区的互补输出。

10.2.10　使用刹车功能

当使用刹车功能时，依据相应的控制位（TIMx_BDTR 寄存器中的 MOE、OSSI 和 OSSR 位，TIMx_CR2 寄存器中的 OISx 和 OISxN 位），输出使能信号和无效电平都会被修改。但无论何时，OCx 和 OCxN 输出不能在同一时间同时处于有效电平上。

刹车源既可以是刹车输入引脚又可以是一个时钟失败事件。时钟失败事件由复位时钟控

制器中的时钟安全系统（CSS）产生。系统复位后，刹车电路被禁止，MOE 位为低。设置 TIMx_BDTR 寄存器中的 BKE 位可以使能刹车功能，刹车输入信号的极性可以通过配置同一个寄存器中的 BKP 位选择。BKE 和 BKP 可以同时被修改。当写入 BKE 和 BKP 位时，在真正写入之前会有 1 个 APB 时钟周期的延迟，因此需要等待一个 APB 时钟周期之后，才能正确地读回写入的位。

因为 MOE 下降沿可以是异步的，在实际信号（作用在输出端）和同步控制位（在 TIMx_BDTR 寄存器中）之间设置了一个再同步电路。这个再同步电路会在异步信号和同步信号之间产生延迟。特别的，如果当它为低时写 MOE=1，则读出它之前必须先插入一个延时（空指令）才能读到正确的值。这是因为写入的是异步信号而读的是同步信号。

当发生刹车时（在刹车输入端出现选定的电平），有下述动作。

- MOE 位被异步地清除，将输出置于无效状态、空闲状态或者复位状态（由 OSSI 位选择）。这个特性在 MCU 的振荡器关闭时依然有效。
- 一旦 MOE=0，每一个输出通道输出由 TIMx_CR2 寄存器中的 OISx 位设定的电平。如果 OSSI=0，则定时器释放使能输出，否则使能输出始终为高。
- 当使用互补输出时，有下述动作。
  - 输出首先被置于复位状态，即无效的状态（取决于极性）。这是异步操作，即使定时器 没有时钟时，此功能也有效。
  - 如果定时器的时钟依然存在，死区生成器将会重新生效。在死区之后根据 OISx 和 OISxN 位指示的电平驱动输出端口。即使在这种情况下，OCx 和 OCxN 也不能被同时驱动到有效的电平。注意，因为重新同步 MOE，死区时间比通常情况下长一些（大约 2 个 ck_tim 的时钟周期）。
  - 如果 OSSI=0，定时器释放使能输出，否则保持使能输出；或一旦 CCxE 与 CCxNE 之一变高时，使能输出变为高。
- 如果设置了 TIMx_DIER 寄存器中的 BIE 位，当刹车状态标志（TIMx_SR 寄存器中的 BIF 位）为 1 时，则产生一个中断。如果设置了 TIMx_DIER 寄存器中的 BDE 位，则产生一个 DMA 请求。
- 如果设置了 TIMx_BDTR 寄存器中的 AOE 位，在下一个更新事件 UEV 时 MOE 位被自动置位。例如，这可以用来进行整型。否则，MOE 始终保持低直到被再次置 1。此时，这个特性可以被用在安全方面，你可以把刹车输入连到电源驱动的报警输出、热敏传感器或者其他安全器件上。

刹车由 BRK 输入产生，它的有效极性是可编程的，且由 TIMx_BDTR 寄存器中的 BKE 位开启。除了刹车输入和输出管理，刹车电路中还实现了写保护以保证应用程序的安全。它允许用户冻结几个配置参数（死区长度，OCx/OCxN 极性和被禁止的状态，OCxM 配置，刹车使能和极性）。用户可以通过 TIMx_BDTR 寄存器中的 LOCK 位，从三级保护中选择一种。在 MCU 复位后，LOCK 位只能被修改一次。

10.2.11 产生六步 PWM 输出

当在一个通道上需要互补输出时，预装载位有 OCxM、CCxE 和 CCxNE。在发生 COM

换相事件时，这些预装载位被传送到影子寄存器位。这样就可以预先设置好下一步骤配置，并在同一个时刻更改所有通道的配置。COM 可以通过设置 TIMx_EGR 寄存器的 COM 位由软件产生，或在 TRGI 上升沿由硬件产生。当发生 COM 事件时会设置一个标志位（TIMx_SR 寄存器中的 COMIF 位），这时如果已设置了 TIMx_DIER 寄存器的 COMIE 位，则产生一个中断；如果已设置了 TIMx_DIER 寄存器的 COMDE 位，则产生一个 DMA 请求。

10.2.12　编码器接口模式

选择编码器接口模式的方法是：如果计数器只在 TI2 的边沿计数，则置 TIMx_SMCR 寄存器中的 SMS=001；如果只在 TI1 边沿计数，则置 SMS=010；如果计数器同时在 TI1 和 TI2 边沿计数，则置 SMS=011。

通过设置 TIMx_CCER 寄存器中的 CC1P 和 CC2P 位，可以选择 TI1 和 TI2 极性；如果需要，还可以对输入滤波器编程。CC1P 和 CC2P 位必须保持为低。

两个输入 TI1 和 TI2 被用来作为增量编码器的接口。假定计数器已经启动（TIMx_CR1 寄存器中的 CEN=1），则计数器由每次在 TI1FP1 或 TI2FP2 上的有效跳变驱动。TI1FP1 和 TI2FP2 是 TI1 和 TI2 在通过输入滤波器和极性控制后的信号；如果没有滤波和变相，则 TI1FP1=TI1，TI2FP2=TI2。根据两个输入信号的跳变顺序，产生了计数脉冲和方向信号。依据两个输入信号的跳变顺序，计数器向上或向下计数，同时硬件对 TIMx_CR1 寄存器的 DIR 位修改。不管计数器是依靠 TI1 还是 TI2 或者同时依靠 TI1 和 TI2 计数，在任一输入端（TI1 或者 TI2）的跳变都会重新计算 DIR 位。

编码器接口模式基本上相当于使用了一个带有方向选择的外部时钟。这意味着计数器只在 0 到 TIMx_ARR 寄存器的自动装载值之间连续计数（根据方向，或是 0 到 ARR 计数，或是 ARR 到 0 计数）。所以在开始计数之前必须配置 TIMx_ARR；同样，捕获器、比较器、预分频器、重复计数器、触发输出特性等仍工作如常。编码器模式和外部时钟模式 2 不兼容，因此不能同时操作。

在这个模式下，计数器依照增量编码器的速度和方向被自动地修改，因此计数器的内容始终指示着编码器的位置。计数方向与相连的传感器旋转的方向对应。表 10-2 列出了所有可能的组合，假设 TI1 和 TI2 不同时变换。

表 10-2　计数方向与编码器信号的关系

有效边沿	相对信号的电平（TI1FP1 对应 TI2，TI2FP2 对应 TI2	TI1FP1 信号		TI2FP2 信号	
		上升	下降	上升	下降
仅在 TI1 计数	高	向下计数	向上计数	不计数	不计数
	低	向上计数	向下计数	不计数	不计数
仅在 TI2 计数	高	不计数	不计数	向上计数	向下计数
	低	不计数	不计数	向下计数	向上计数
在 TI1 和 TI2 上计数	高	向下计数	向上计数	向上计数	向下计数
	低	向上计数	向下计数	向下计数	向上计数

一个外部的增量编码器可以直接与 MCU 连接而不需要外部接口逻辑。但是，一般会使用比较器将编码器的差动输出转换到数字信号，这大大增加了抗噪声干扰能力。编码器输出的第三个信号表示机械零点，可以把它连接到一个外部中断输入并触发一个计数器复位。

图 10-12 是一个计数器操作的实例，显示了计数信号的产生和方向控制。它还显示了当选择了双边沿时，输入抖动是如何被抑制的，抖动可能会在传感器的位置靠近一个转换点时产生。在这个例子中，我们假定配置如下：

- CC1S=01（TIMx_CCMR1 寄存器，IC1FP1 映射到 TI1）
- CC2S=01（TIMx_CCMR2 寄存器，IC2FP2 映射到 TI2）
- CC1P=0（TIMx_CCER 寄存器，IC1FP1 不反相，IC1FP1=TI1）
- CC2P=0（TIMx_CCER 寄存器，IC2FP2 不反相，IC2FP2=TI2）
- SMS=011（TIMx_SMCR 寄存器，所有的输入均在上升沿和下降沿有效）
- CEN=1（TIMx_CR1 寄存器，计数器使能）

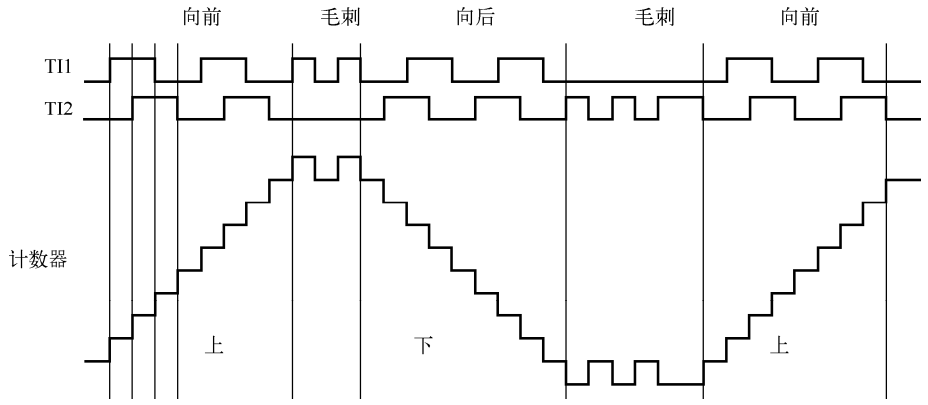


图 10-12　编码器模式下的计数器操作实例

当定时器配置成编码器接口模式时，用于指示传感器当前位置，配合使用第二个定时器配置为捕获模式，通过测量两个编码器事件的间隔，从而获得动态信息（速度、加速度、减速度）。编码器输出还可用来指示机械零点。根据两个事件间的间隔，可以按照固定的时间读出计数器。如果有必要，可以把计数器的值锁存到第三个输入捕获寄存器（由另一个定时器产生的捕获信号必须是周期性），也可以通过一个由实时时钟产生的 DMA 请求来读取它的值。

10.3　固　件　库

定时器驱动的函数（见表 10-3）分成 8 部分，其中时基管理部分主要配置定时器的时基础单元，设置/获取预分频，设置/获取预自动加载，配置定时器模式。

表 10-3　定时器固件库

类　别	函　数	功 能 描 述
时基管理函数	void　TIM_ARRPreloadConfig (TIM_TypeDef *TIMx, FunctionalState NewState)	使能或禁用在 ARR 上预装载寄存器
	void　TIM_Cmd (TIM_TypeDef *TIMx, FunctionalState NewState)	使能或禁用指定的 TIM 外设
	void　TIM_CounterModeConfig (TIM_TypeDef *TIMx, uint16_t TIM_CounterMode)	指定 TIMx 计数器模式
	void　TIM_DeInit (TIM_TypeDef *TIMx)	复位 TIMx 外设寄存器到默认复位值
	uint32_t　TIM_GetCounter (TIM_TypeDef *TIMx)	读取 TIMx 计数值
	uint16_t　TIM_GetPrescaler (TIM_TypeDef *TIMx)	获取 TIMx 预分频值
	void　TIM_PrescalerConfig (TIM_TyeDef *TIMx, uint16_t Prescaler, uint16_ t TIM_PSCReloadMode)	配置 TIMx 分频器
	void　TIM_SelectOnePulseMode (TIM_TypeDef *TIMx, uint16_t TIM_OPMode)	选择 TIMx 的单脉冲模式
	void　TIM_SetAutoreload (TIM_TypeDef *TIMx, uint32_t Autoreload)	设置 TIMx 的自动加载寄存器值
	void　TIM_SetClockDivision (TIM_TypeDef *TIMx, uint16_t TIM_CKD)	设置 TIMx 时钟除数
	void　TIM_SetCounter (TIM_TypeDef *TIMx, uint32_t Counter)	设置 TIMx 计数器寄存器值
	void　TIM_TimeBaseInit (TIM_TypeDef *TIMx, TIM_TimeBaseInit TypeDef * TIM_TimeBaseInitStruct)	根据输入参数初始化 TIMx 外设
	void　TIM_TimeBaseStructInit (TIM_TimeBaseInitTypeDef *TIM_TimeBaseInitStruct)	将 TIM_TimeBaseInitStruct 的成员设置成默认值
	void　TIM_UpdateDisableConfig (TIM_TypeDef *TIMx, FunctionalState NewState)	使能或禁用 TIMx 更新事件
	void　TIM_UpdateRequestConfig (TIM_TypeDef *TIMx, uint16_t TIM_UpdateSource)	配置 TIMx 更新请求中断源
高级控制定时器	void　TIM_BDTRConfig (TIM_TypeDef *TIMx, TIM_BDTRInitTypeDef *TIM_BDTRInitStruct)	配置刹车、死区、锁电平、OSSI/OSSR 状态以及 AOE（自动输出使能）
	void　TIM_BDTRStructInit (TIM_BDTRInitTypeDef *TIM_BDTRInitStruct)	默认值填充 TIM_BDRTInitStruct 成员
	void　TIM_CtrlPWMOutputs (TIM_TypeDef *TIMx, FunctionalState NewState)	使能或禁用 TIM 外设主输出
输出比较器管理功能	void　TIM_CCPreloadControl (TIM_TypeDef *TIMx, FunctionalState NewState)	设置或者复位 TIM 外设捕获比较预装载控制位
	void　TIM_CCxCmd (TIM_TypeDef *TIMx, uint16_t TIM_Channel, uint16_t TIM_CCx)	使能或禁用 TIM 捕获通道 x
	void　TIM_CCxNCmd (TIM_TypeDef *TIMx, uint16_t TIM_Channel, uint16_t TIM_CCxN)	使能或禁用 TIM 捕获比较通道 xN
	void　TIM_ClearOC1Ref (TIM_TypeDef *TIMx, uint16_t TIM_OCClear)	清除或保护 OEREF1 信号
	void　TIM_ClearOC2Ref (TIM_TypeDef *TIMx, uint16_t TIM_OCClear)	清除或保护 OCREF2 信号
	void　TIM_ClearOC3Ref (TIM_TypeDef *TIMx, uint16_t TIM_OCClear)	清除或保护 OCREF3 信号
	void　TIM_ClearOC4Ref (TIM_TypeDef *TIMx, uint16_t TIM_OCClear)	清除或保护 OCREF4 信号
	void　TIM_ForcedOC1Config (TIM_TypeDef *TIMx, uint16_t TIM_ForcedAction)	强制 TIMx 输出 1 波形为高或低电平
	void　TIM_OC1FastConfig (TIM_TypeDef *TIMx, uint16_t TIM_OCFast)	配置 TIMx 输出比较器 1 快速模式
	void　TIM_OC1Init (TIM_TypeDef *TIMx, TIM_OCInitTypeDef *TIM_OCInitStruct)	由输入参数初始化 TIMx 通道 1
	void　TIM_OC1NPolarityConfig (TIM_TypeDef *TIMx, uint16_t TIM_OCNPolarity)	配置 TIMx 通道 1N 极性
	void　TIM_OC1PolarityConfig (TIM_TypeDef *TIMx, uint16_t TIM_OCPolarity)	配置 TIMx 通道极性
	void　TIM_OC1PreloadConfig (TIM_TypeDef *TIMx, uint16_t TIM_OCPreload)	使能或禁用 TIMx 的 CCR1 中外设预加载寄存器

续表

类　别	函　数	功 能 描 述
输出比较器管理功能	Void TIM_SelectCOM (TIM_TypeDef *TIMx, FunctionalState NewState)	选择 TIM 外设换向事件
	void TIM_SelectOCREFClear (TIM_TypeDef *TIMx, uint16_t TIM_OCReferenceClear)	选择 OCReference 清除源
	void TIM_SelectOCxM (TIM_TypeDef *TIMx, uint16_t TIM_Channel, uint16_t TIM_OCMode)	选择 TIM 输出比较模式
	void TIM_SetCompare1 (TIM_TypeDef *TIMx, uint32_t Compare1)	设置 TIMx 捕获比较寄存器值
输入捕获管理函数	uint32_t TIM_GetCapture1 (TIM_TypeDef *TIMx)	读取输入捕获 1 的值
	void TIM_ICInit (TIM_TypeDef *TIMx, TIM_ICInitTypeDef *TIM_ICInitStruct)	由输入参数初始化 TIM 外设
	void TIM_ICStructInit (TIM_ICInitTypeDef *TIM_ICInitStruct)	设置 TIM_ICInitStruct 成员初始值
	void TIM_PWMIConfig (TIM_TypeDef *TIMx, TIM_ICInitTypeDef *TIM_ICInitStruct)	为测量外部 PWM 信号，由输入参数配置 TIM 外设
	void TIM_SetIC1Prescaler (TIM_TypeDef *TIMx, uint16_t TIM_ICPSC)	设置 TIMx 输入捕获 1 预分频值
中断、DMA 和标志管理函数	void TIM_ClearFlag (TIM_TypeDef *TIMx, uint16_t TIM_FLAG)	清除 TIMx 的挂起标志
	void TIM_ClearITPendingBit (TIM_TypeDef *TIMx, uint16_t TIM_IT)	清除 TIMx 的中断挂起位
	void TIM_DMACmd (TIM_TypeDef *TIMx, uint16_t TIM_DMASource, FunctionalState NewState)	使能或禁用 TIMx 的 DMA 请求
	void TIM_DMAConfig (TIM_TypeDef *TIMx, uint16_t TIM_DMABase, uint16_t TIM_DMABurstLength)	配置 TIMx 的 DMA 接口
	void TIM_GenerateEvent (TIM_TypeDef *TIMx, uint16_t TIM_EventSource)	配置 TIMx 的由软件生成事件
	FlagStatus TIM_GetFlagStatus (TIM_TypeDef *TIMx, uint16_t TIM_FLAG)	检查 TIM 的特定标志是否设置
	ITStatus TIM_GetITStatus (TIM_TypeDef *TIMx, uint16_t TIM_IT)	检查 TIM 中断是否发生
	void TIM_ITConfig (TIM_TypeDef *TIMx, uint16_t TIM_IT, FunctionalState NewState)	使能或禁用指定的 TIM 中断
	void TIM_SelectCCDMA (TIM_TypeDef *TIMx, FunctionalState NewState)	选择 TIMx 外设捕获比较 DMa 源
时钟管理功能	void TIM_ETRClockMode1Config (TIM_TypeDef *TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter)	配置外部时钟模式
	void TIM_ETRClockMode2Config (TIM_TypeDef *TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter)	配置外部时钟模式 2
	void TIM_InternalClockConfig (TIM_TypeDef *TIMx)	配置 TIMx 内部时钟
	void TIM_ITRxExternalClockConfig (TIM_TypeDef *TIMx, uint16_t TIM_InputTriggerSource)	配置 TIMx 内部触发器为外部时钟
	void TIM_TIxExternalClockConfig (TIM_TypeDef *TIMx, uint16_t TIM_TIxExternalCLKSource, uint16_t TIM_ICPolarity, uint16_t ICFilter)	配置 TIMx 触发器为外部时钟
同步	void TIM_ETRConfig (TIM_TypeDef *TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter)	配置 TIMx 外部触发器
	void TIM_SelectInputTrigger (TIM_TypeDef *TIMx, uint16_t TIM_InputTriggerSource)	选择输入触发源
	void TIM_SelectMasterSlaveMode (TIM_TypeDef *TIMx, uint16_t TIM_ MasterSlaveMode)	设置或复位 TIMx 主/从模式
	void TIM_SelectOutputTrigger (TIM_TypeDef *TIMx, uint16_t TIM_TRGOSource)	选择 TIMx 触发器输出模式
	void TIM_SelectSlaveMode (TIM_TypeDef *TIMx, uint16_t TIM_SlaveMode)	选择 TIMx 从模式
特定接口管理	void TIM_EncoderInterfaceConfig (TIM_TypeDef *TIMx, uint16_t TIM_EncoderMode, uint16_t TIM_IC1Polarity, uint16_t TIM_IC2Polarity)	配置 TIMx 解码器接口
	void TIM_SelectHallSensor (TIM_TypeDef *TIMx, FunctionalState NewState)	使能或禁用 TIMx 的霍尔传感器接口
特定的重映射管理	void TIM_RemapConfig (TIM_TypeDef *TIMx, uint16_t TIM_Remap)	配置 TIM14 重映射输入容量

将定时器的用法根据功能划分成时基用法、刹车、输出比较、输入捕获、同步。

（1）时基模式的用法

- 使用 RCC_APBxPeriphClockCmd(RCC_APBxPeriph_TIMx, ENABLE)使能 TIM 时钟。
- 设置 TIM_TimeBaseInitStruct 变量。
- 调用 TIM_TimeBaseInit(TIMx, &TIM_TimeBaseInitStruct)配置时基单元。
- 使能 NVIC。
- 使用 TIM_ITConfig(TIMx, TIM_IT_Update)使能相应的中断。
- 调用 TIM_Cmd(ENABLE)函数使用 TIM 计数器。

（2）刹车功能的用法

- 在相关的输出比较模式中配置完定时器通道后。
- 设置 TIM_BDTRInitStruct 的刹车极性、死区时间、锁级别、OSSI/OSSR 状态和 AOE（自动输出使能）。
- 调用 TIM_BDTRConfig(TIMx, &TIM_BDTRInitStruct)配置定时器。
- 使用 TIM_CtrlPWMOutputs(TIM1, ENABLE)使能主输出。
- 一旦刹车发生，定时器输出信号被置于复位状态或者确定状态（根据 TIM_BDTRConfig()函数的配置）。

（3）输出比较模式

- 使用 RCC_APBxPeriphClockCmd(RCC_APBxPeriph_TIMx, ENABLE)使能 TIM 时钟。
- 通过配置相应的 GPIO 引脚配置 TIM 引脚。
- 配置时基，否则定时器将以默认值运行。
  - 自动加载值：0xFFFF
  - 预分频值：0x0000
  - 计数器模式：向上计数
  - 时钟分频因子：TIM_CKD_DIV1
- 使用下列参数设置 TIM_OCInitStruct。
  - TIM 输出比较模式：TIM_OCMode
  - TIM 输出状态：TIM_OutputState
  - TIM 脉冲值：TIM_Pulse
  - TIM 输出比较极性：TIM_OCPolarity
- 调用 TIM_OCxInit(TIMx, &TIM_OCInitStruct)配置相关通道。
- 调用 TIM_CMD（ENABLE）使能 TIM 计数器。
- PWM 模式下，必须调用 TIM_OCxPreloadConfig(TIMx, TIM_OCPreload_ENABLE)。
- 如需要使用中断或者 DMA，需使能 NVIC（或 DMA），使用 TIM_ITConfig(TIMx, TIM_IT_CCx)（或 TIM_DMA_Cmd(TIMx, TIM_DMA_CCx)）使能中断或 DMA。

（4）输入捕获模式的用法

- 使用 RCC_APBxPeriphClockCmd(RCC_APBxPeriph_TIMx, ENABLE)使能 TIM 时钟。
- 通过配置相应的 GPIO 引脚配置 TIM 引脚。
- 如不配置时基单元，将使用默认配置。
- 使用如下参数配置 TIM_ICInitStruct。
  - TIM 通道：TIM_Channel

- TIM 输入捕获极性：TIM_ICPolarity
- TIM 输入捕获选择：TIM_ICSelection
- TIM 输入捕获预分频：TIM_ICPrescaler
- TIM 输入捕获滤波值：TIM_ICFilter

❑ 调用 TIM_ICInit(TIMx, &TIM_ICInitStruct)配置相关通道和测量输入信号的占空比或者频率，或者调用 TIM_PWMIConfig(TIMx, &TIM_ICInitStruct) 配置相关通道。

❑ 为了读取测量频率使能 NVIC 或者 DMA。

❑ 使用 TIM_ITConfig(TIMx, TIM_IT_CCx)或者 TIM_DMA_Cmd(TIMx, TIM_DMA_CCx)使能中断，读取测量值。

❑ 调用 TIM_Cmd(ENABLE)使用 TIM 计数器。

❑ 使用 TIM_GetCapturex(TIMx)读取捕获值。

（5）同步模式用法

① 两个或者多个定时器情况。

❑ 配置主定时器的相关函数。

- void TIM_SelectOutputTrigger(TIM_TypeDef* TIMx, uint16_t TIM_TRGOSource)
- void TIM_SelectMasterSlaveMode(TIM_TypeDef* TIMx, uint16_t TIM_MasterSlaveMode)

❑ 配置从定时器的相关函数。

- void TIM_SelectInputTrigger(TIM_TypeDef* TIMx,uint16_t TIM_InputTriggerSource)
- void TIM_SelectSlaveMode(TIM_TypeDef* TIMx, uint16_t TIM_SlaveMode)

② 定时器和外部触发（ETR 引脚）。

❑ 配置外部触发函数。

void TIM_ETRConfig(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter)

❑ 配置从定时器。

- void TIM_SelectInputTrigger(TIM_TypeDef* TIMx, uint16_t TIM_InputTriggerSource)
- void TIM_SelectSlaveMode(TIM_TypeDef* TIMx, uint16_t TIM_SlaveMode)

10.4 SPWM 实例

SWPM 技术是采用等幅不等宽的 PWM 来替代正弦波。SPWM 规则采样法在工程中被大量使用。SPWM 规则采样法是采用三角波作为载波的规则采样法，下面公式为计算脉冲宽度公式。

$$\delta = \frac{T_c}{2}(1 + \alpha \sin(w_r T_D))$$

根据公式，使用 Matlab 写出如下程序，并将计算得到的结果以 C 语言数组定义的形式输出到 sintable.c 文件中，将输出的文件复制到本章的 MDK 工程中即可。下面的 Matlab 代码是以 10kHz 为载波，调制波为 50Hz 为例进行说明。

```
clear;clc;
fc=10*1000 %载波
fr=50 %调制波
a=1 %调制比
tc=1/fc
n =fc/fr

for i =0:n-1
    delta(i+1)=(tc/2 )*(1+a*sin(2 *3.14*fr*((i+1)*tc+tc/4))) %  计算出来的时间
end
SystemCoreClock=48000000 %CPU 时钟
TimerPeriod = (SystemCoreClock / fc ) - 1; %  计算预定表的值，也就是多少个时钟计数为一个周期
delta1= fc*delta ;%换算成占空比
Channel1Pulse= delta1 * TimerPeriod ;   % CCR1 跳转值，占空比为 delta1
file ='c:\sintable.c'%直接通过 Matlab 创建一个文件供 MDK 调用
fid=fopen(file,'wt')
fprintf(fid,'int sintable[%d]={',n)
for i=1:n
    fprintf(fid,'%7.f',Channel1Pulse(i))
    if (i<n)
        fprintf(fid,',')
    end
end
fprintf(fid,'};');
fclose (fid)
```

STM32F0x 的程序部分采用中断更新占空比。下面是定时器 1 的配置部分，PA0.8 为对应 SPWM 引脚输出，PA0.9 一直输出占空比为 50%的 10kHz 方波。

```
void NVIC_Configuration(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;
  NVIC_InitStructure.NVIC_IRQChannel = TIM1_BRK_UP_TRG_COM_IRQn;//TIM1_CC_IRQn;
//TIM1 中断
  NVIC_InitStructure.NVIC_IRQChannelPriority = 0x00;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}

void TIM_Config(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;
  /*  使能 GPIO 时钟  */
  RCC_AHBPeriphClockCmd( RCC_AHBPeriph_GPIOA, ENABLE);
  /*  配置 GPIO 管脚复用*/
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 ;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
```

```
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP ;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource8, GPIO_AF_2);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource9, GPIO_AF_2);
}

void TIM_PWM_Config(void)
{
  /*计算预定表的值，多少个时钟计数为一个周期*/
  TimerPeriod = (SystemCoreClock / 10000 ) - 1;
  Channel1Pulse =sintable[0];
 /*计算 CCR2 跳转值，在占空比为 50%时*/
  Channel2Pulse = (uint16_t) (((uint32_t) 500 * (TimerPeriod - 1)) / 1000);
  /* TIM1 时钟使能 */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1 , ENABLE);
  /* Time 定时基础设置*/
  TIM_TimeBaseStructure.TIM_Prescaler = 0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;  /* Time 定时设置为向上
计算模式*/
  TIM_TimeBaseStructure.TIM_Period = TimerPeriod;
  TIM_TimeBaseStructure.TIM_ClockDivision = 0;
  TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);
  /* 频道 1 的 PWM 模式设置 */
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStructure.TIM_OutputNState = TIM_OutputNState_Enable;
  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
  TIM_OCInitStructure.TIM_OCNPolarity = TIM_OCNPolarity_High;
  TIM_OCInitStructure.TIM_OCIdleState = TIM_OCIdleState_Set;
  TIM_OCInitStructure.TIM_OCNIdleState = TIM_OCIdleState_Reset;

  TIM_OCInitStructure.TIM_Pulse = Channel1Pulse;//使能通道 1 配置
  TIM_OC1Init(TIM1, &TIM_OCInitStructure);
   TIM_OCInitStructure.TIM_Pulse = Channel2Pulse;//使能通道 2 配置
  TIM_OC2Init(TIM1, &TIM_OCInitStructure);
  /* TIM1 计算器使能*/
   TIM_ITConfig(TIM1,TIM_IT_Update,ENABLE ); //使能指定的 TIM1 中断，允许更新中断
  TIM_Cmd(TIM1, ENABLE);/* TIM1 主输出使能 */

   TIM_CtrlPWMOutputs(TIM1, ENABLE);
   NVIC_Configuration() ;
   }
```

中断部分程序为 TIM1_BRK_UP_TRG_COM_IRQHandler。需要注意的是，定时器 1 对

应两个中断向量。

```
/*
    *定时器 1 中断服务程序
*/
void TIM1_BRK_UP_TRG_COM_IRQHandler(void )
{
    if (TIM_GetITStatus(TIM1, TIM_IT_Update) != RESET)   //检查 TIM1 更新中断发生与否
            {
                    spwm();//更新占空比
                    TIM_ClearITPendingBit(TIM1, TIM_IT_Update);
            }
}
    void spwm (void)
{
    if (counter ==sizeof( sintable ) / sizeof( sintable[0] ))counter=0;
    TIM_SetCompare1(TIM1,sintable[counter]);//在每个中断中更新占空比
    counter++;

}
```

说明：本节的例程需要外配 8MHz 晶振，程序中需倍频到 48MHz。由于 ST 的 Discovery 无外部晶振，不能用 Discovery 做实验。

在将 SPWM 用于驱动电力元件之前，可通过 RC 滤波器验证正弦波的情况。图 10-13 是可观察本节例程的 RC 电路取值。由于载波的频率不同，R、C 取值可能不同，如果仅仅观察验证正弦波效果，R、C 往大取值即可。

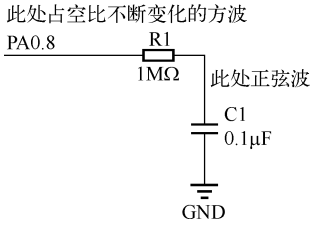


图 10-13　RC 滤波

10.5　小　　结

本章通过定时器 1 介绍了定时器的时基、比较输出、捕获输入等特征。由于 PWM 技术在现代逆变电路中被广泛使用并数字化，所以定时器最重要的应用场合是电力电子行业。在 10.4 节给出了 SPWM 的实例，给出如何计算 SPWM 占空比的代码，采用通过中断更新 SPWM 的占空比的方式。读者可自行实现通过 DMA 方式更新占空比。

第11章

模数转换器（ADC）

STM32F0 有一个 12 位逐次逼近型 ADC（SAR ADC），它有 19 个通道，可测量 16 个外部和 3 个内部信号源。各通道的 A/D 转换以单次、连续、扫描或间断模式执行。ADC 的结果以左对齐或右对齐方式存储在 16 位数据寄存器中。模拟看门狗用于检测输入电压是否超限，可用于峰值检测（在电源设计产品中经常用到峰值检测功能）。

11.1 ADC 主要特性

STM32F0x 的 ADC 主要特性如下。

（1）高性能。

- 12 位、10 位、8 位或 6 位可配置分辨率。
- ADC 转换时间：1.0μs@12 位分辨率（1MHz）、0.93μs@10 位分辨率、在更低的转换分辨率下可达到更快的转换时间。
- 自校准。
- 可编程采样时间。
- 带内嵌数据一致性的数据对齐。
- DMA 支持。

（2）低功耗。

- 降低 PLCK 频率仍保持最佳的 ADC 性能。例如：无论在何种 PCLK 的频率下，保持 1.0μs 的 ADC 转换时间。
- 等待模式：运行在 PLCK 低速下，防止 ADC 过度采样。
- 自动关闭模式：ADC 除了在转换期间工作外，其他时间 ADC 自动断电。该方式大大降低了 ADC 的功耗。

（3）模拟输入通道。

- 从外部 GPIO 口连接的 16 通道模拟输入。
- 1 通道内部温度传感（V_{SENSE}）输入。
- 1 通道的内部参考电压（V_{REFINT}）输入。
- 1 通道的外部电池 VBAT 供电引脚输入。

（4）多种启动转换方式。

- 由软件触发。
- 由硬件触发（从 TIM1、TIM2、TIM3 和 TIM15 发出的内部定时器事件）。

（5）转换模式。

- 可转换单通道或一序列通道。
- 每个触发以单独模式触发选定通道。
- 连续方式转换选定输入通道。
- 间断模式（Discontinuous mode）。

（6）转换完成后、序列转换完成、模拟看门狗或转换溢出事件都可以产生中断。

（7）模拟看门狗。

（8）ADC 供电要求：2.4～3.6V。

（9）ADC 输入范围：VSSA≤VIN≤VDDA。

注意：在 STM32F1 系列中无内部通道，STM32F0 包含了内部温度传感（VSENSE）输入。STM32F030 系列无电池检测功能。另外 STM32F030F4（STM32F030 中的 TSSOP20 封装）无模拟地，做硬件设计时需注意。

11.2　ADC 功能描述

STM32F0x 的 ADC 是 SAR 的 ADC。图 11-1 是 SAR 的 ADC 基本原理图。模拟输入电压（VIN）由采样/保持电路保持。比较器与 SAR 逻辑采用二进制搜索算法将 V_{IN} 电压值与不断调整的 V_{DAC} 输出值比较，获得模拟输入的数字量形式，N 位转换结果储存在寄存器内。

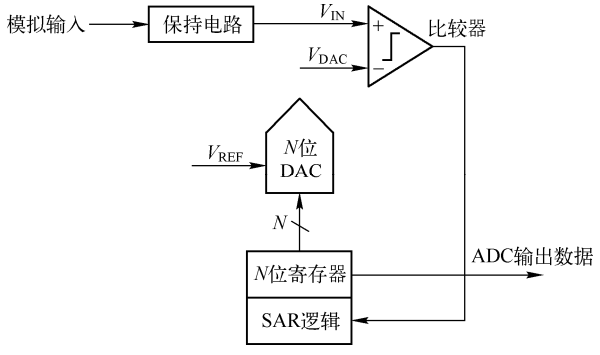


图 11-1　SAR 的 ADC 原理图

图 11-2 是 STM32F0x 的 ADC 原理框图。ADC_IN[15:0]代表 16 路模拟输入。TIMx_TRG 从定时器来的内部信息，作为触发源；V_{REF} 是内部参考电压的输出；V_{BAT} 是 V_{BAT} 引脚输入电压。

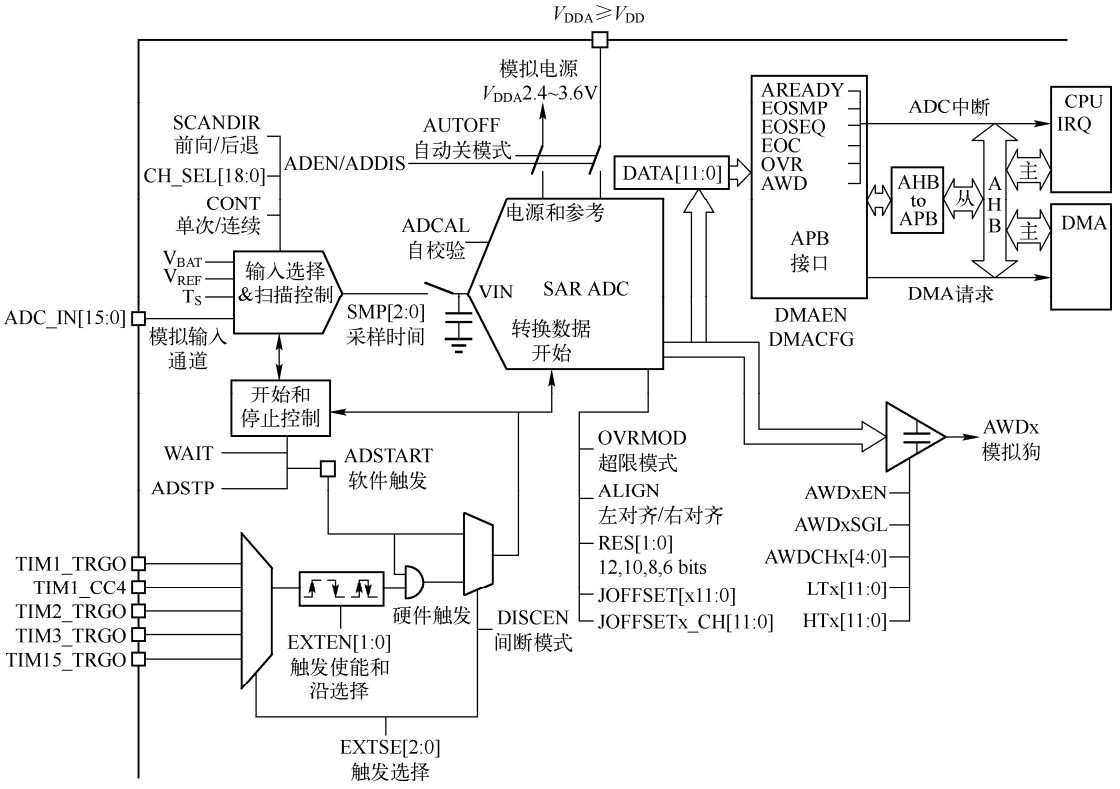


图 11-2　STM32F0x 的 ADC 模块图

11.2.1　校准

ADC 本身具有校准功能，在校准期间，ADC 计算一个用于 ADC 校准的 7 位校准因子（ADC 断电后丢失）。在 A/D 转换前应执行校准操作，校准用于消除各芯片 AD 转换的偏移误差。在 ADC 校准期间和校准未完成前，不能使用 ADC 模块。

校准是由软件设置 ADCAL=1 来实现初始化的。其只能在 ADC 禁用（ADEN=0）时完成初始化。在校准期间，ADCAL 位必须保持为 1。当校准完成后，ADCAL 被硬件清 0。校准完成后，可从 ADC_DR 寄存器的 6:0 位读出校准因子。

当 ADC 禁用时（ADEN=0）校准因子保持原值。然而，若 ADC 长时间禁用，建议在启用 ADC 之前重新做一次 ADC 校准操作。校准因子在 ADC 每次断电后丢失（例如当器件从待机模式恢复或由 VBAT 供电）。校准因子可以保留到 ADC 模块掉电；ADC 模块未使能时，也可保留；进入待机或电池供电模式，ADC 模块掉电，不再保留。

固件库中的 ADC_GetCalibrationFactor 函数实现校准功能。

11.2.2　ADC 开关控制

默认情况下，禁用 ADC 模块且处于断电模式（ADEN=0），在 ADC 开始精确转换前需要一个稳定时间 t_{STAB}。两个控制位用于开启或关闭 ADC：设置 ADEN=1，开启 ADC。当 ADC 模块准备好时 ADRDY 标志置 1；设置 ADDIS=1 来关闭 ADC，并使 ADC 处于断电模

式。当 ADC 模块全关断后，硬件自动清除 ADEN 和 ADDIS 位。

ADC 转换也可由设置 SWSTART=1 来启动或由外部触发事件来触发启动。下列是开启 ADC 的过程：

① 在 ADC_CR 寄存器中设置 ADEN=1。

② 等待直到 ADC_ISR 寄存器中的 ADRDY=1（当 ADC 启动后 ADRDY 置位）。若 ADRDYIE=1（ADC_IER 寄存器中）设置 ADC 准备好的中断使能时，也可用中断来处理 ADC 准备好。

以下为禁用 ADC 的过程：

① 检查 ADC_CR 寄存器中的 ADSTART 是否为 0 以确保 ADC 不在转换过程中。若需要，可对 ADC_CR 寄存器中的 ADSTP 置 1 来停止正在进行的 ADC 转换，并等待 ADSTP 被硬件清 0（清 0 表示转换停止完成）。

② 设置 ADC_CR 寄存器中的 ADDIS=1。

③ 若应用要求，则可等待 ADC_CR 寄存器中的 ADEN 位为 0，其表明 ADC 模块完全关闭（一旦 ADEN=0 时 ADDIS 自动清 0）。

注：在自动关闭模式（AUTOFF=1），电源开关由硬件自动执行并且 ADRDY 标志不变。

11.2.3　ADC 时钟

ADC 时钟（ADC_CLK）独立于 APB 时钟（PCLK）。ADC_CLK 时钟源可由 RCC 寄存器选用专用的 14MHz 内部振荡器或者 PCLK 时钟/2（/4）（最大不能超过 14MHz 的 ADC_CLK），如图 11-3 所示。表 11-1 是两种时钟源对比。

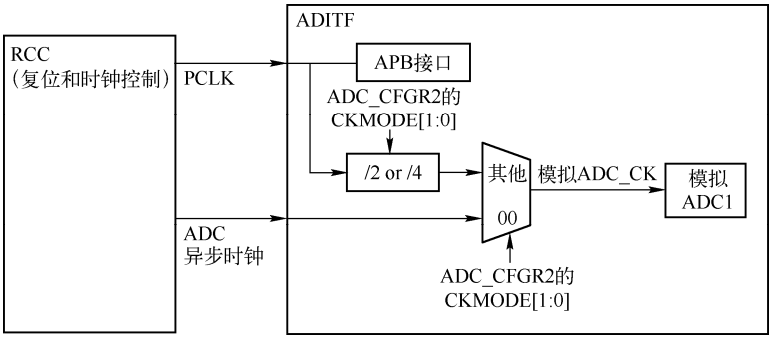


图 11-3　ADC 时钟方案

表 11-1　ADC 时钟源对比

ADC 时钟源	优　点	缺　点
专用的片上 14MHz RC 振荡器	无论 MCU 的运行频率如何，都可以保证最高的 ADC 工作频率。 可以使用自动节电模式（自动开启或关闭 14MHz 的内部振荡器）	触发信号的同步会带来抖动。 触发事件和转换的起始时刻之间的延迟不确定
APB 时钟的 2 分频或 4 分频（最高 14MHz）	不会有时钟域之间的同步带来的抖动。 触发事件和转换的起始时刻之间的延迟是确定的，从而保证转换之间的时间间隔是固定的	ADC 的转换时间与系统时钟频率相关

11.2.4 ADC 配置

ADC 配置时需要满足一些条件：软件必须在 ADC 禁止（ADEN 必须为 0）的情况下改写 ADC_CR 寄存器中的 ADCAL 和 ADEN 位；软件必须在 ADC 开启且没有关闭请求挂起（即 ADEN=1 且 ADDIS=0）的情况下改写 ADC_CR 寄存器中的 ADSTART 和 ADDIS 位；对于 ADC_IER、ADC_CFGRi、ADC_SMPR、ADC_TR、ADC_CHSELR 和 ADC_CCR 寄存器中的控制位，软件必须在 ADC 开启（ADEN =1）且无转换期间（ADSTART = 0）的情况下才能进行改写；软件必须在 ADC 开启且无挂起请求（ADSTART = 1 和 ADDIS = 0）的情况下改写 ADC_CR 寄存器中的 ADSTP 位。

注：没有硬件保护机制去防止软件修改以上所述操作规则。假如出现了上述约定，ADC 会进入一种不确定状态。为了从这种情况下恢复，ADC 必须被关闭（ADEN=0 且 ADC_CR 寄存器清 0）。

11.2.5 通道选择

共有 19 路复用通道：16 个从 GPIO 引脚引入的模拟输入（ADC_IN0,…,ADC_IN15），3 个内部模拟输入（温度传感、内部参考电压、VBAT 通道）。

ADC 可以转换一个单一通道或自动扫描一个序列通道。被转换的通道序列必须在通道选择寄存器 ADC_CHSELR 中编程选择：每个模拟输入通道有专门的一位选择位（CHSEL0,…,CHSEL18）。

ADC 扫描的通道顺序由 ADC_CFGR1 中 SCANDIR 位的配置来决定。

- SCANDIR=0。向前扫描：从通道 0 到通道 18。
- SCANDIR=1。回退扫描：从通道 18 到通道 0。

温度传感器连接到 ADC_IN16，内部参考电压连接到 ADC_IN17，V_{BAT} 连接到 ADC_IN18（STM32F030 系列无此功能）。

11.2.6 转换模式

转换模式分为单次转换、连续转换模式两种方式。

单次转换模式下，ADC 执行一次序列转换，转换所有被选的通道。当 ADC_CFGR1 寄存器中的 CONT=0 时，ADC 为单次转换模式。ADC 转换可由 ADC_CR 寄存器中设置 ADSTART 位或者硬件触发事件启动。在序列通道的转换中，每次通道转换完成后：

- 转换的数据结果存放到 16 位寄存器 ADC_DR 中；
- EOC（转换结束标志）标志置位；
- 若 EOCIE 位置位则产生一个中断。

通道序列转换完成后：

- EOS（序列结束）标志置位；
- 若 EOSIE 位置位则产生一个中断。

转换结束后，ADC 停止直到新的触发事件或 ADSTART 重新置位。

注：若转换单一通道，则可编程一个长度为 1 的一个转换序列。

在连续转换模式中，当软件或硬件触发事件产生，ADC 执行一个序列转换，一次性转换序列中的所有通道而后自动重新执行该序列的下一次转换。当寄存器 ADC_CFGR1 中的 CONT=1 时，ADC 选择为连续转换模式。ADC 转换可由 ADC_CR 寄存器中设置 ADSTART 或者硬件触发事件启动。

在序列通道的转换中，每次转换完成后：

- 转换的数据结果存放到 16 位寄存器 ADC_DR 中；
- EOC（转换结束标志）标志置位；
- 若 EOCIE 位置位则产生一个中断。

通道序列转换完成后：

- EOS（序列结束）标志置位；
- 若 EOSIE 位置位则产生一个中断。

一次序列转换结束后，ADC 立即重新转换相同的序列通道。

注：①若转换单一通道，则可编程一个长度为 1 的一个转换序列。

② 让 ADC 同时处于断续转换模式和连续转换模式是不可能的事情。在这种情况下（DISCEN=1，CONT=1），其表现为连续模式禁止（即为单次转换模）。

11.2.7　启动与停止转换

软件用设置 ADSTART=1 启动 ADC 转换。当 ADSTART 置 1，如果 EXTEN 为 0（软件触发）时，立即开始转换，否则在下一个所选择的活动边沿硬件触发。ADSTART 位也用于说明目前 ADC 转换操作是否正在进行。当 ADC 处于空闲时，该位可重新配置为 ADSTART=0。下面情况下 ADSTART 位由硬件清除：

- 单次转换模式由软件触发（CONT=0，EXTSEL=0x0），在序列转换结束后（EOS=1）。
- 在所有的情况下（CONT=X，EXTSEL=X），在软件调用并执行 ADSTP 过程后。

注：① 在连续模式（CONT=1）下，ADSTART 位不能由 EOS 引发的硬件清除，其原因是自动重新开始序列转换。

② 当硬件触发选择为单次转换模式（CONT=0 和 EXTSEL ≠ 0x00），则当 EOS 标志设置后，ADSTART 不会被硬件清 0。这就避免了需要软件重新设置 ADSTART 位且要确保无硬件触发事件错过。

用软件设置 ADC_CR 寄存器中的 ADSTP=1 可以停止当前正在进行的转换。中止会复位 ADC 的操作并让 ADC 进入空闲状态，为下次转换作好准备。当 ADSTP 由软件设置为 1，任何当前的转换中止且转换结果丢弃（ADC_DR 寄存器不用当前的转换值进行更新）。扫描序列也被中止并复位（即重新启动 ADC 时会用新的序列进行转换）。一旦结束该过程，ADSTP 和 ADSTART 位都由硬件清 0。

11.3 外部触发和触发极性

一次转换或一个序列的转换可由软件或外部事件（例如：定时器捕获、输入引脚）触发。若 EXTEN[1:0]≠ “0b00”，则外部事件在其所选择的极性上可以用于触发转换。当软件设置 ADSTART=1 时，触发选择有效。当正在进行 ADC 转换时，任何硬件触发都会被忽略。当 ADSTART=0 时，任何硬件触发都会忽略。表 11-2 给出 EXTEN[1:0]值与其相对应的极性。

表 11-2 配置触发极性

源	EXTEN[1:0]
触发检测禁止	00
在上升沿时检测	01
在下降沿时检测	10
在上升沿及下降沿都检测	11

EXTSEL[2:0]控制位用于选择可触发转换的事件。表 11-3 给出了规则转换可能的外部触发。软件源触发事件可由设置 ADC_CR 寄存器中的 ADSTART 位来产生。

表 11-3 外部触发

Name	触发源	类型	EXTSEL[2:0]
EXT0	TIM1_TRGO	片内定时器产生的内部信号	000
EXT1	TIM1_CC4		001
EXT2	TIM3_TRGO		010
EXT3	TIM1_TRGO		011
EXT4	TIM15_TRGO		100
EXT5	保留		101
EXT6	保留		110
EXT7	保留	外部引脚	111

注：在转换时外部触发极性不能改变。

该模式由设置 ADC_CFGR1 寄存器中的 DISCEN 位来开启。在这个模式（DISCEN=1）下，需要硬件或软件的触发事件去启动定义在一个序列中的每次转换。相反，DISCEN=0 时，一个硬件或软件的触发事件就可以启动定义在一个序列中的所有转换。

例如：

- DISCEN=1，需要转换的通道为 0、3、7、10。
  - 1st 触发：通道 0 被转换且一个 EOC 事件产生。
  - 2nd 触发：通道 3 被转换且一个 EOC 事件产生。
  - 3rd 触发：通道 7 被转换且一个 EOC 事件产生。
  - 4th 触发：通道 10 被转换且产生 EOC 和 EOS 事件。
  - 5th 触发：通道 0 被转换且一个 EOC 事件产生。

- 6th 触发：通道 3 被转换且一个 EOC 事件产生。

- DISCEN=0，需要转换的通道为 0、3、7、10。
  - 1st 触发：整个完整的序列转换，依次为通道 0、3、7 和 10。每次转换产生一个 EOC 事件，到最后一通道还产生一个 EOS 事件。
  - 任何后续的触发事件会重启整个序列转换。

注：让 ADC 同时处于断续转换模式和连续转换模式是不可能的事情。在这种情况下（DISCEN=1，CONT=1），其表现为连续模式禁止。

通过降低转换分辨率来获取更快的转换时间（tSAR）是可行的。转换分辨率可通过设置 ADC_CFGR1 寄存器中的 RES[1:0]来配置为 12、10、8 或 6 位模式。当应用不需要高精度数据时，可用低的转换分辨率来加快转换时间。转换结果也是 12 位宽度且低位补 0。低分辨率模式减少逐次逼近的转换时间。

ADC 通知应用每次转换结束（EOC）事件。一旦在 ADC_DR 寄存器中的一个转换数据有效后，ADC 在 ADC_ISR 寄存器中设置 EOC 标志表明转换完成。当 ADC_IER 中的 EOCIE 置为 1 时，则会产生一个 EOC 中断。EOC 标志由软件写 1 清除或读 ADC_DR 寄存器来清除。ADC 同样在 ADC_ISR 寄存器中给出采样阶段结束标志 EOSMP。EOSMP 标志可写 1 清除。当在 ADC_IER 寄存器中的 EOSMPIE 置为 1 后，则会产生一个 EOSMP 中断。

ADC 通知应用每次序列转换结束（EOS）事件。一旦一个转换序列的最后一个通道转换数据有效后，ADC 在 ADC_ISR 寄存器中设置 EOS 标志。当 ADC_IER 中的 EOSIE 位置 1 时，则会产生 EOS 中断。EOS 标志由软件写 1 清 0。

11.4　数 据 管 理

在每次转换结束（当 EOC 事件产生时），转换的结果数据被存放到在 16 位宽 ADC_DR 数据寄存器中。ADC_DR 数据格式与所配置的数据对齐和转换分辨率有关。ADC_CFR1 寄存器中的 ALIGN 位用于选择数据存储的对齐方式，数据可选为右对齐（ALIGN=0）或左对齐（ALIGN=1）。

新的转换数据完成，但上一数据未被 CPU 或者 DMA 读走，为 ADC 过度采样。若 EOC 为 1 的情况下，新的转换完成，那么 CPU 就会在 ADC_ISR 寄存器中的 OVR 标志被置位，表明 ADC 采样过度。当 ADC_IER 寄存器中的 OVRIE 置位时，产生一个 ADC OVR 中断。当过度采样事件发生时，ADC 会继续操作并且继续转换，除非软件决定停止并复位这个序列转换，可用软件设置 ADC_CR 寄存器中的 ADSTP 为 1 来停止 ADC 转换。OVR 标志可用软件写 1 清除。

通过设置 ADC_CFGR1 寄存器的 OVRMOD 位决定过度采样时，数据被保留还是被覆盖。

- 若 OVRMOD=0，表明一个过度采样事件保持数据寄存器的值防止被覆盖，即丢新保旧。若 OVR 保持为 1，则后续的转换会被执行，但结果都被丢弃。
- 若 OVRMOD=1，表明数据寄存器用最后的转换结果覆盖但先前未读的数据。若 OVR 保持为 1，则后续的转换被执行，且 ADC_DR 寄存器存放着最后转换的结果值。

若 ADC 的转换足够慢，转换序列可由软件来控制。这种情况下，软件应用 EOC 标志及其关联的中断去处理每个转换数据。当每次转换结束时，在 ADC_ISR 寄存器中的 EOC 位置位，此时可读 ADC_DR 寄存器的转换值。ADC_CFGR1 寄存器中的 OVRMOD 位可配为 0 来管理过度采样事件。

OVRMOD 位置 1 并忽略 OVR 标志，则转换一个或多个通道不用每次将转换结果读取。当 OVRMOD=1 时，过采样不能阻止 ADC 继续转换，并且 ADC_DR 寄存器中的数据一直为最后转换的数据。

因为所有通道的转换结果数据存放到一个单一的数据寄存器中，故当转换通道超过 1 个时用 DMA 方式会更有效。这样可以避免丢失 ADC_DR 寄存器中的转换结果。当 DMA 模式开启时（ADC_CFGR1 寄存器中的 DMAEN=1），每次转换结束时都会产生一个 DMA 请求。这样就允许把在 ADC_DR 寄存器中的转换数据传送到软件指定的目标地址中。尽管如此，如果因 DMA 未响应 DMA 请求服务而出现过冲（OVR=1），ADC 停止产生 DMA 请求，DMA 也出传输新转换数据（当 OVR=0 时，会继续传输）。这也可以认为所有传输到 RAM 中的数据都是有效的（因无效的数据再也不传输了）。根据 OVRMOD 位的配置，ADC_DR 寄存器中的数据可选择为保持或覆盖。DMA 传输请求会被阻止直到软件清除 OVR 位。有两种不同的 DMA 模式，取决于 ADC_CFGR1 寄存器中的 DMACFG 位的配置。

- DMA 一次模式（one shot mode）（DMACFG=0）：当 DMA 编程用于传输固定长度的数据时，可选用该模式。
- DMA 循环模式（DMACFG=1）：当 DMA 编程为循环模式或双缓冲模式时，可选用该模式。

DMA 一次模式，ADC 在每次转换数据完产生 DMA 请求，一旦 DMA 到达最后一个 DMA 传输停止生成 DMA 请求。当 DMA 传输完成（配置在 DMA 控制器中的所有传输已经完成）：

- ADC 数据寄存器的内容冻结。
- 任何进行中的转换中止且结果值丢弃。
- 不给 DMA 控制器发出新的 DMA 请求。假如仍有 ADC 转换启动，这种方式可避免产生一个 ADC 过冲错误。
- ADC 扫描序列停止并复位。
- DMA 停止。

DMA 循环模式（DMACFG=1）时，即使 DMA 达到最后一个 DMA 的传输，ADC 也会在每次转换的数据有效时产生一次 DMA 请求。这允许 DMA 配置为循环模式来处理连续模拟输入数据流。

11.5 低功耗特性

降低功耗通过等待模式和自动方式实现。

等待模式可用于在低速运行时简化软件以及优化应用程序的性能，但容易产生 ADC 过度采样的情况。当 ADC_CFGR [AUTDLY]为 1 时，一个新的转换只有在刚才的 ADC 数据处

理完后（比如 ADC_DR 寄存器中的数据被读取或 EOC 标志已被清除）才开始。这是一种自适应 ADC 速度和自适应系统读取 ADC 数据速度的方法。

注：当正在转换中或自动延迟产生的情况下，任一硬件产生的触发都会被忽略。

通过 ADC_CFGR1 寄存器中的 AUTOFF 置 1，开启 ADC 的自动关闭功能，降低功耗。当 AUTOFF=1 时，无转换时 ADC 断电；当转换启动（由软件或硬件触发）时 ADC 自动唤醒。一个启动时间在触发开始转换与采样之间自动插入。一旦序列转换结束后 ADC 自动关闭，大大降低应用的功耗，适用于需要相对较少转换或转换请求的时间间隔足够长（如低频的硬件触发）的应用。在低频场合，自动关闭模式可与等待（AUTDLY=1）转换模式组合使用。如果在等待时 ADC 自动关闭，ADC_DR 被读走后立即启动 ADC，可大大省电。

11.6　ADC 中断

ADC 中断可由以下任一事件产生：

- ADC 上电，当 ADC 准备好（ADRDY 标志）；
- 任何一次的转换结束（EOC 标志）；
- 序列转换结束（EOS 标志）；
- 当模拟看门狗检测发生（AWD 标志）；
- 当采样阶段结束发生（EOSMP 标志）；
- 当数据过度采样发生（OVR 标志）。

中断的使能位以及中断标志如图 11-4 所示，固件库中 ADC_ITConfig 函数用于使能 ADC 中断，ADC_ClearFlag 用于清除挂起标志。

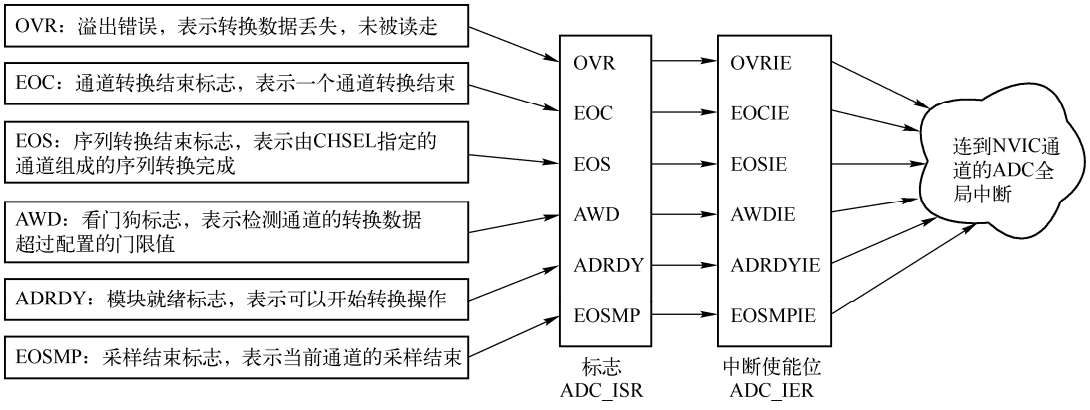


图 11-4　ADC 中断

11.7　ADC 固件库

stm32f0xx_adc.c 文件包含了 ADC 的驱动函数，涉及初始化和配置、省电、模拟看门狗

配置、温度传感器、Vrefint（内部参考电压）和 Vbat（电源电池）管理、ADC 通道配置、ADC 通道 DMA 配置、中断和标志管理。ADC 的驱动函数见表 11-4。

表 11-4　ADC 的驱动函数

void ADC_AnalogWatchdogCmd (ADC_TypeDef *ADCx, FunctionalState NewState)	使能或者禁用模拟看门狗
void ADC_AnalogWatchdogSingleChannelCmd (ADC_TypeDef *ADCx, FunctionalState NewState)	使能或者禁用模拟看门狗单通道
void ADC_AnalogWatchdogSingleChannelConfig (ADC_TypeDef *ADCx, uint32_t ADC_AnalogWatchdog_Channel)	配置监测单通道的模拟看门狗
void ADC_AnalogWatchdogThresholdsConfig (ADC_TypeDef *ADCx, uint16_t HighThreshold, uint16_t LowThreshold)	配置模拟看门狗的高低门限值
void ADC_AutoPowerOffCmd (ADC_TypeDef *ADCx, FunctionalState NewState)	使能或禁用 ADC 省电模式
void ADC_ChannelConfig (ADC_TypeDef *ADCx, uint32_t ADC_Channel, uint32_t ADC_SampleTime)	配置 ADC 和采样时间
void ADC_ClearFlag (ADC_TypeDef *ADCx, uint32_t ADC_FLAG)	清除 ADC 的挂起标志
void ADC_ClearITPendingBit (ADC_TypeDef *ADCx, uint32_t ADC_IT)	轻质 ADCx 的挂起标志
void ADC_ClockModeConfig (ADC_TypeDef *ADCx, uint32_t ADC_ClockMode)	使用异步时钟（专用 14MHz 时钟）或者来源 ADC 总线的 APB 时钟同步时钟配置 ADC
void ADC_Cmd (ADC_TypeDef *ADCx, FunctionalState NewState)	使能或禁用指定 ADC 外设
void ADC_ContinuousModeCmd (ADC_TypeDef *ADCx, FunctionalState NewState)	使能选定 ADCx 通道的连续模式
void ADC_DeInit (ADC_TypeDef *ADCx)	重新初始化 ADC 外设寄存器为默认复位值
void ADC_DiscModeCmd (ADC_TypeDef *ADCx, FunctionalState NewState)	使能选定 ADCx 通道的非连续模式
void ADC_DMACmd (ADC_TypeDef *ADCx, FunctionalState NewState)	使能或禁用选定 ADC 的 DMA 请求
void ADC_DMARequestModeConfig (ADC_TypeDef *ADCx, uint32_t ADC_DMARequestMode)	最后一次传递后使能或禁用 ADC 的 DMA 请求
uint32_t ADC_GetCalibrationFactor (ADC_TypeDef *ADCx)	激活选定 ADC 的校准操作
uint16_t ADC_GetConversionValue (ADC_TypeDef *ADCx)	返回 ADCx 转换结果
FlagStatus ADC_GetFlagStatus (ADC_TypeDef *ADCx, uint32_t ADC_ FLAG)	检查 ADC 标志是否置位
ITStatus ADC_GetITStatus (ADC_TypeDef *ADCx, uint32_t ADC_IT)	检查 ADC 中断是否发证
void ADC_Init (ADC_TypeDef *ADCx, ADC_InitTypeDef *ADC_InitStruct)	初始化 ADCx 外设
void ADC_ITConfig (ADC_TypeDef *ADCx, uint32_t ADC_IT, FunctionalState NewState)	使能或禁用指定 ADC 中断
void ADC_JitterCmd (ADC_TypeDef *ADCx, uint32_t ADC_JitterOff, FunctionalState NewState)	使能或禁用抖动
void ADC_OverrunModeCmd (ADC_TypeDef *ADCx, FunctionalState NewState)	使用选定 ADC 通道超限转换
void ADC_StartOfConversion (ADC_TypeDef *ADCx)	启动选定的 ADC 通道转换
void ADC_StopOfConversion (ADC_TypeDef *ADCx)	停止正在进行的 ADC 转换
void ADC_StructInit (ADC_InitTypeDef *ADC_InitStruct)	用默认值田中 ADC_InitStruct
void ADC_TempSensorCmd (FunctionalState NewState)	使用或禁用温度传感器通道
void ADC_VbatCmd (FunctionalState NewState)	使用或禁用 Vbat 通道
void ADC_VrefintCmd (FunctionalState NewState)	使用或禁用 Vrefint 通道
void ADC_WaitModeCmd (ADC_TypeDef *ADCx, FunctionalState NewState)	使能或禁用等待转换模式

使用 ADC 驱动函数的方法分为配置启动 ADC、ADC 通道组配置和 DMA 配置。

使用 RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1，ENABLE)使能 ADC 时钟。

（1）ADC 引脚配置

- 使用 RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOx，ENABLE)函数使能 ADC 的 GPIO 时钟；
- 使用 GPIO_Init()配置引脚为模拟方式；
- 使用 ADC_Init()函数配置 ADC 转换分辨率、数据对齐、外部触发和沿、扫描方向和使能/禁用连续模式；
- 使用 ADC_Cmd 函数激活 ADC 外设。

（2）ADC 通道组配置

- 使用 ADC_Init()和 ADC_ChannelConfig()函数配置 ADC 通道特征；
- 使用 ADC_DiscModeCmd()函数激活非连续模式；
- 使用 ADC_OverrunModeCmd()函数激活超限模式；
- 使用 ADC_GetCalibrationFactor()函数激活校准模式；
- 使用 ADC_GetConversionValue()函数读取 ADC 转换数据。

（3）ADC 通道的 DMA 配置

- 使用 ADC_DMACmd()函数使能 ADC 通道的 DMA 模式；
- 使用 ADC_DMARequestModeConfig()函数配置 DMA 传送请求。

11.8　STM32F05x(07x)的 DAC 与比较器

在 STM32F05X 与 STM32F07x 另有两个模拟器件：DAC 与比较器。

DAC 模块是 12 位的电压输出数模转换器，可配置成 8 位或者 12 位，可配置 DMA 使用。DAC 在 12 位模式下，数据可以是左对齐或者右对齐；输入参考电压、VDDA（与 ADC 共用）可用。DAC 的框图如图 11-5 所示。图 11-5 中的触发选择器旁边的记录信号为 5 路 DAC 触发源。

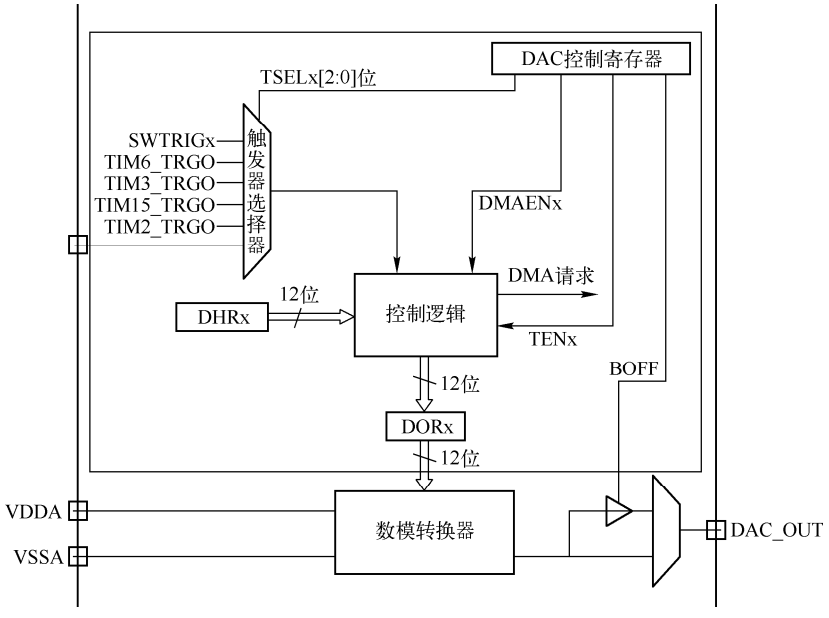


图 11-5　DAC 原理框图

注：由于 STM32F030x4、STM32F030x6、STM32F030x8 芯片无 DAC 功能。在需要 DAC 功能情况下，可使用芯片的 PWM 与 RC 滤波器（或其他形式滤波器电路）配合生成 DAC 信号。

STM32F05x 与 STM32F07x 包含两个通用比较器，用作独立外设（在 IO 上对应端子均可用）或者与定时器结合，可用于：

- 低电压下的模拟信号唤醒功能；
- 模拟信号检测；
- 与 DAC 和定时器输出的 PWM 相结合，组成逐周期的电流控制回路。

图 11-6 是比较器的框图，每个比较器的门限值可以为 3 个 I/O 引脚、DAC1、内部电压和三个等分电压值（1/4、1/2、3/4）。输出端可以重定向到一个 I/O 端口或多个定时器输入端，可以触发以下事件：

- 捕获事件；
- OCref_clr 事件（逐周期电流控制）；
- 为实现快速 PWM 关断的中断事件。

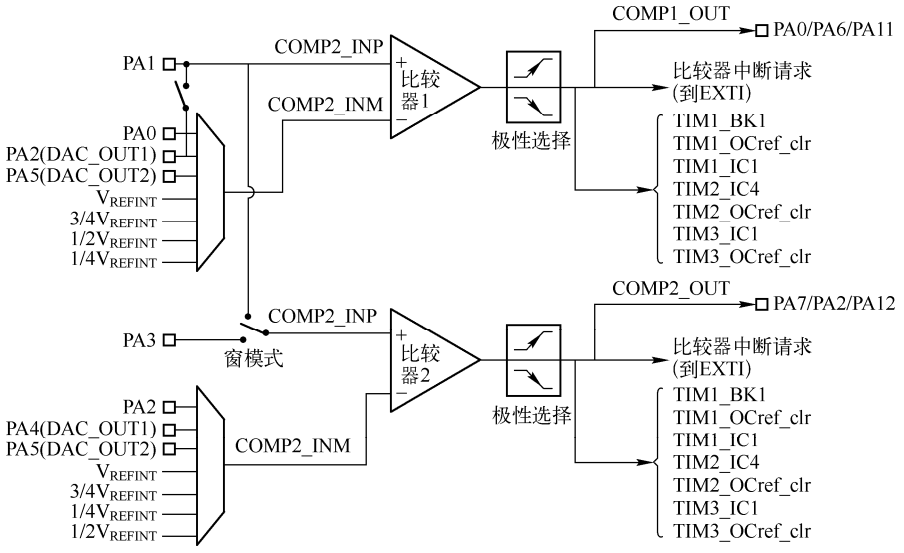


图 11-6　比较器原理框图

两个比较器可以组合在一个窗口比较器中使用。每个比较器都可产生中断，并支持从睡眠和停止模式唤醒（通过 EXTI 控制器）。

既然已经有 ADC 功能了，为何 STM32F0x 还有比较器呢?由于 ADC 需要 1.5 mA 典型电流损耗，会降低电池使用时间，所以低功耗场合推荐使用模拟比较器方式用于输入电压比较。

11.9　USB 电压监测

目前手机几乎都是通过 5V 充电器或者计算机输出的 USB 数据线进行充电的。虽然很多产品都是标称 5V 充电器，但由于各种原因，一些充电器可达 6V。本例采用 STM32F030 芯片监测

5V 电压并通过 PL2303 HXD 的串口将采样值发送给 PC。由于 STM32F0 的 AD 通道是 3.3V 耐压。5V 输入电压采用千分之一精度的 10kΩ电阻进行分压，AD 通道采集分压电压值。

由于仅一个 AD 通道，对采集速度无要求，所以未采用 DMA 方式。但由于循环监视状态方式不适合产品使用，所以采用定时器 1 触发 AD 采集。采集完毕产生中断，在中断服务程序中读取采样结果的方式。下列代码中 ADC_Config()中的 ADC_InitStructure. ADC_ContinuousConvMode 设置成 DISABLE（见 11.2.6 节中的连续转换说明）。由于采用了定时器，故 ADC_ExternalTrigConv 设置成 ADC_ExternalTrigConv_ T1_CC4（见 11.3 节中的 ADC 触发）。由于 STM32F0 家族中 ADC 中断定义有所不同，故采用宏定义方式区分中断号（见 Starup_Stm32F0xxx.s 文件）。

```
static void ADC_Config(void)
{
  ADC_InitTypeDef              ADC_InitStructure;
  GPIO_InitTypeDef             GPIO_InitStructure;
  /* GPIOA 时钟使能 */
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
  /* ADC 时钟使能*/
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 ; /*配置 PA0.1 */
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN; //模拟输入
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
  GPIO_Init(GPIOA, &GPIO_InitStructure);
  /* ADC1 恢复初始状态 */
  ADC_DeInit(ADC1);
  /* 初始化 ADC 结构体 */
  ADC_StructInit(&ADC_InitStructure);
  /*配置 ADC1 分辨率为 12 位 */
  ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
  ADC_InitStructure.ADC_ContinuousConvMode = DISABLE ;//每次转换均有时钟触发，不采用连续转换方式
  ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_Rising;//外部触发
  ADC_InitStructure.ADC_ExternalTrigConv =   ADC_ExternalTrigConv_T1_CC4;//T1 触发
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;//12 位数据右对齐
  ADC_InitStructure.ADC_ScanDirection = ADC_ScanDirection_Upward;
  ADC_Init(ADC1, &ADC_InitStructure);

  /* 配置采样时间 */
  ADC_ChannelConfig(ADC1, ADC_Channel_1 , ADC_SampleTime_239_5Cycles);
  /* ADC 校准 */
  ADC_GetCalibrationFactor(ADC1);
  /* 使能 ADC 外设 */
  ADC_Cmd(ADC1, ENABLE);
  /*等待 ADRDY 标志 */
  while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_ADRDY));
   //中断配置
```

```
    //ADC_ITConfig(ADC1,ADC_IT_EOSEQ,ENABLE );// 仅一个通道，序列转换结束中断不需要
    ADC_ITConfig(ADC1,ADC_IT_EOC,ENABLE);//转换结束中断
    #ifdef  STM32F030 //由于 STM32F030 系列的中断向量定义不同，具体可参见 stm32f0xx.h 文件定义。
    NVIC_InitStructure.NVIC_IRQChannel = ADC1_IRQn;
    #else
        NVIC_InitStructure.NVIC_IRQChannel = ADC1_COMP_IRQn;
    #endif
   NVIC_InitStructure.NVIC_IRQChannelPriority = 0x01;
   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
   NVIC_Init(&NVIC_InitStructure);
   /* ADC1 开始转换 */
  ADC_StartOfConversion(ADC1);
}
```

ADC 对应的中断程序如下。

```
 #ifdef STM32F030   //由于启动文件中对中断服务程序的定义名称不同，详见 Starup_Stm32F0xxx.s 文件
void ADC1_IRQHandler (void)
#else
void ADC1_COMP_IRQHandler(void)
#endif
{
    if (ADC_GetITStatus(ADC1, ADC_FLAG_EOC)!= RESET) //转换结束中断
    {
        endflag=1;
    /*读取 ADC1 转换数据 */
    ADC1ConvertedValue =ADC_GetConversionValue(ADC1);
        ADC_ClearITPendingBit(ADC1, ADC_FLAG_EOC);
      }
        if (ADC_GetITStatus(ADC1, ADC_FLAG_EOSEQ)!= RESET) //序列中断
    { ADC_ClearITPendingBit(ADC1, ADC_IT_EOSEQ);
    }

}
```

11.10 小　　结

本章主要涉及 ADC 时钟源、如何进行转换、转换的启动停止、ADC 触发源和采样数据的管理。最后给出了通过外部触发源（定时器 1）触发 ADC 采集，以及 ADC 采样结束产生中断，在中断服务程序读取 ADC 采样数据方式的实例。在 11.4 数据管理一节说明 ADC 采样数据除了中断方式，还有 DMA 方式。在 12 章介绍 DMA 原理后，将给出通过 DMA 方式读取采样数据的实例。

第12章

DMA 控制

直接存储器存取（DMA）的作用是在无需微控器干预情况下，实现外设和存储器之间或者存储器和存储器之间的高速数据传输，以为其他操作保留了微控器资源。STM32F0 的两个 DMA 控制器共有 5 个通道与一个用来协调各个 DMA 请求优先权的仲裁器。每个通道专门用来管理来自于一个或多个外设对存储器的访问请求。

12.1 DMA 主要特性

STM32F0x 的 DMA 具有如下特点：

- 5 个独立的可配置通道（请求）。
- 每个通道都直接连接专用的硬件 DMA 请求，每个通道都同样支持软件触发。这些配置通过软件来完成。
- 在同一个 DMA 模块上，多个请求间的优先权可以通过软件编程设置（共有四级：很高、高、中等和低），优先权设置相等时由硬件决定请求 1 优先于请求 2，依此类推。
- 数据源和目标数据区的传输宽度可不一样（字节、半字、全字），并模拟了打包和拆包的过程。源和目标地址必须按数据传输宽度对齐。
- 支持循环的缓冲器管理。
- 每个通道都有 3 个事件标志（DMA 半传输、DMA 传输完成和 DMA 传输出错），这 3 个事件标志逻辑或成单独的中断请求。
- 存储器和存储器间的传输。
- 外设到存储器和存储器到外设，外设到外设间的传输。
- 闪存、SRAM、APB 和 AHB 外设均可作为访问的源和目标。
- 可编程的数据传输数目最大为 65536。

12.2 DMA 功能描述

12.2.1 DMA 原理

Cortex-M0 DMA 原理框图如图 12-1 所示，DMA 控制器和 Cortex-M0 内核共享系统数据

总线，执行直接存储器数据传输。当 CPU 和 DMA 同时访问相同的目标（RAM 或外设）时，DMA 请求会暂停 CPU 访问系统总线达若干个周期，总线仲裁器执行循环调度，以确保 CPU 至少可以得到一半的系统总线带宽（存储器和外设），仲裁器根据优先级管理着通道的请求和启动外设/存储器的访问优先级管理。

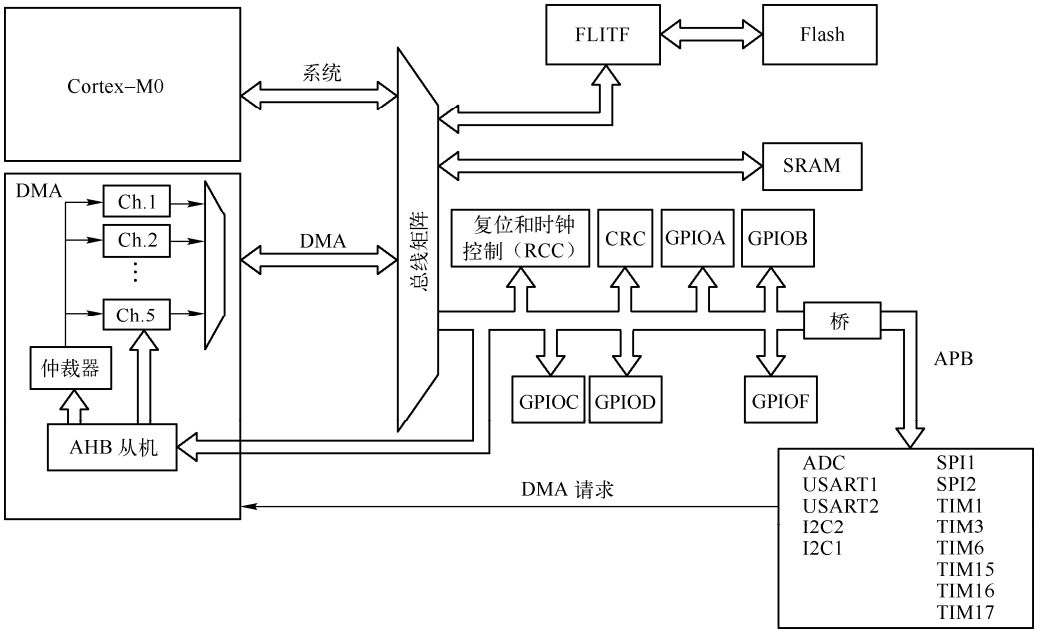


图 12-1　Cortex-M0 DMA 原理框图

在发生一个事件后，外设向 DMA 控制器发送一个请求信号。DMA 控制器根据通道的优先权处理请求。当 DMA 控制器开始访问发出请求的外设时，DMA 控制器立即发送给它一个应答信号。当从 DMA 控制器得到应答信号时，外设立即释放它的请求。一旦外设释放了这个请求，DMA 控制器同时撤销应答信号。如果有更多的请求时，外设可以启动下一个周期。总之，每次的 DMA 传输由 3 组操作组成：

- 从外设数据寄存器或者从当前外设/存储器地址寄存器指示的存储器地址取数据，第一次传输时的开始地址是 DMA_CPARx 或 DMA_CMARx 寄存器指定的外设基地址或存储器单元。
- 从外设数据寄存器或者从当前外设/存储器地址寄存器指示的存储器地址取数据，第一次传输时的开始地址是 DMA_CPARx 或 DMA_CMARx 寄存器指定的外设基地址或存储器单元。
- 对 DMA_CNDTRx 寄存器执行一次递减操作，DMA_CNDTRx 寄存器存放着未完成的 DMA 操作的计数。

每个通道都可以在外设寄存器固定地址和存储器地址之间执行 DMA 传输。DMA 传输的数据量是可编程的，最大达到 65 535。每次传输之后相应的计数寄存器都做一次递减操作，直到计数为 0。外设和存储器的传输数据量可以通过 DMA_CCRx 寄存器中的 PSIZE 和 MSIZE 位编程。通过设置 DMA_CCRx 寄存器中的 PINC 和 MINC 标志位，外设和存储器的

指针在每次传输后可以有选择地完成自动增量。当设置为增量模式时，下一个要传输的地址将是前一个地址加上增量值，增量值取决于所选的数据宽度为 1、2 或 4。第一个传输的地址是存放在 DMA_CPARx/DMA_CMARx 寄存器中的地址。在传输过程中，这些寄存器保持它们初始的数值，软件不能改变和读出当前正在传输的地址（它在内部的当前外设/存储器地址寄存器中）。

当通道配置为非循环传输模式时，传输结束后（即传输计数已递减到 0）将不再产生 DMA 请求。为了在 DMA_CNDTRx 寄存器中重新写入传输数目，需要先关闭相应的 DMA 通道。

注：假如一个 DMA 通道关闭，DMA 寄存器的值未复位，DMA 通道寄存器（DMA_CCRx、DMA_CPARx 和 DMA_CMARx）在通道配置期间会保持着初始值。

在循环模式下，最后一次传输结束时，DMA_CNDTRx 寄存器自动重新装载为初始编程的计数值。当前内部地址寄存器从 DMA_CPARx/DMA_CMARx 寄存器中装载基地址值。

1．通道配置过程

下面给出的是 DMA 通道 x（x 代表通道号）的配置步骤。

- 在 DMA_CPARx 寄存器中设置外设寄存器地址。发生外设数据传输时，这个地址将是数据传输的源或目的地址。
- 在 DMA_CMARx 寄存器中设置存储器地址。发生外设数据传输时，传输的数据将从这个地址读出或写入。
- 在 DMA_CNDTRx 寄存器中写入需要传输的数据量，每次 DMA 传输后，该寄存器值递减。
- 在 DMA_CCRx 的 PL[1:0]位中配置通道的优先级。
- 在 DMA_CCRx 中配置数据的传输方向、循环模式、外设和存储器的增量模式、外设和存储器的数据宽度，以及传输一半产生中断或传输完成产生中断的设置。
- 设置 DMA_CCRx 寄存器中的 ENABLE 位来启动该通道。

一旦启动了 DMA 通道，它就可响应连接到该道道上的外设的 DMA 请求。当传输一半的数据后，半传输标志（HTIF）被置 1，当设置了允许半传输中断位（HTIE）时，将产生一个半传输完成的中断请求。当传输完成时，传输完成标志（TCIF）被置 1，当设置了允许传输完成中断位（TCIE）时，将产生一个传输完成的中断请求。

2．循环模式

循环模式用于处理循环缓冲区和连续的数据传输（如 ADC 的扫描模式）。在 DMA_CCRx 寄存器中的 CIRC 位用于开启这一功能。当启动了循环模式，一组的数据传输完成时，计数寄存器将会自动地被恢复成配置该通道时设置的初值，DMA 操作将会继续进行。

3．存储器到存储器模式

DMA 通道的操作可以在没有外设请求的情况下进行，这种操作就是存储器到存储器模式。当设置了 DMA_CCRx 寄存器中的 MEM2MEM 位之后，在软件设置了 DMA_CCRx 寄存器中的 EN 位启动 DMA 通道时，DMA 传输将马上开始。当 DMA_CNDTRx 寄存器变为 0 时，DMA 传输结束。存储器到存储器模式不能与循环模式同时使用。

12.2.2 可编程的数据宽度、数据对齐方式和数据大小端

当 PSIZE 和 MSIZE 不相同时，DMA 模块受数据长和大小端影响。

当 DMA 模块开始一个 AHB 的字节或半字写操作时，数据将在 HWDATA[31:0]总线中未使用的部分重复。因此，如果 DMA 以字节或半字写入不支持字节或半字写操作的 AHB 设备时（即 HSIZE 不适于该模块）不会发生错误，DMA 将按照下面两个例子写入 32 位 HWDATA 数据。

- 当 HSIZE= 半字时，写入半字 0xABCD，DMA 将设置 HWDATA 总线为 0xABCDABCD。
- 当 HSIZE= 字节时，写入字节 0xAB，DMA 将设置 HWDATA 总线为 0xABABABAB。

假定 AHB/APB 桥是一个 AHB 的 32 位从设备，它不处理 HSIZE 参数，它将按照下述方式把任何 AHB 上的字节或半字按 32 位传送到 APB 上。

- 一个 AHB 上对地址 0x0（或 0x1、0x2 或 0x3）的写字节数据 0xB0 操作，将转换到 APB 上对地址 0x0 的写字数据 0xB0B0B0B0 操作。
- 一个 AHB 上对地址 0x0（或 0x2）的写半字数据 0xB1B0 操作，将转换到 APB 上对地址 0x0 的写字数据 0xB1B0B1B0 操作。

例如，如果要写入 APB 后备寄存器（与 32 位地址对齐的 16 位寄存器），需要配置存储器数据源宽度（MSIZE）为 16 位，外设目标数据宽度（PSIZE）为 32 位。

12.2.3 错误管理

读/写一个保留的地址区域，将会产生 DMA 传输错误。当在 DMA 读/写操作时发生 DMA 传输错误时，硬件会自动地清除发生错误的通道所对应的通道配置寄存器（DMA_CCRx）的 EN 位，该通道操作被停止。此时，在 DMA_IFR 寄存器中对应该通道的传输错误中断标志位（TEIF）将被置位，如果在 DMA_CCRx 寄存器中设置了传输错误中断允许位，则将产生中断。

12.2.4 中断

每个 DMA 通道都可以在 DMA 传输过半、传输完成和传输错误时产生中断。为应用的灵活性考虑，通过设置寄存器的不同位来打开这些中断。表 12-1 是 DMA 中断请求。

表 12-1 DMA 中断请求

中断事件	事件标志位	使能控制位
传输过半	HTIF	HTIE
传输完成	TCIF	TCIE
传输错误	TEIF	TEIE

12.2.5 DMA 请求映射

外设（TIMx、ADC、DAC、SPI、I^2C 和 USARTx）的硬件请求简单地进行逻辑或运算

后进入 DMA。这就意味着同一刻只允许一个 DMA 请求进入 DMA 控制器。外设的 DMA 请求，可以通过设置相应外设寄存器中的控制位被独立地开启或关闭。图 12-2 是 STM32F030X4/X6/X8 对应 DMA 请求映射。STM32F0 家族中的其他芯片因外设不同，DMA 映射关系稍微不同，主要差别在一些增加的外设上。

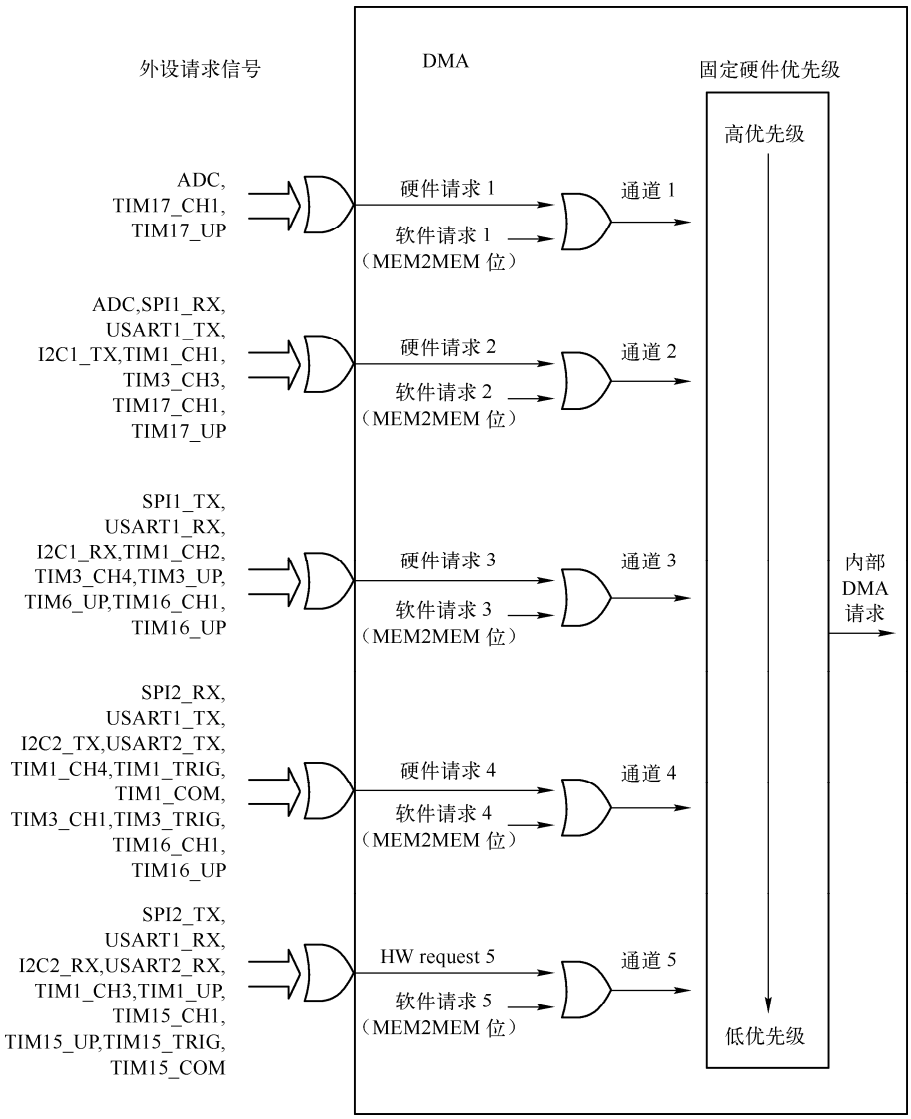


图 12-2　STM32F030X4/X6/X8 对应 DMA 请求映射

12.3　固　件　库

表 12-2 是 DMA 有关的固件库函数。

表 12-2　DMA 固件库

函　　数	说　　明
void　DMA_ClearFlag (uint32_t DMA_FLAG)	清除 DMA 通道的挂起标志
void　DMA_ClearITPendingBit (uint32_t DMA_IT)	清除 DMA 通道的中断挂起标志
void　DMA_Cmd (DMA_Channel_TypeDef *DMAy_Channelx，FunctionalState NewState)	使能或禁用特定的 DMA 通道
void　DMA_DeInit (DMA_Channel_TypeDef *DMAy_Channelx)	复位 DMA 通道寄存器到默认复位值
uint16_t　DMA_GetCurrDataCounter (DMA_Channel_TypeDef *DMAy_Channelx)	返回当前 DMA 通道的剩余数据值
FlagStatus　DMA_GetFlagStatus (uint32_t DMA_FLAG)	检查指定的 DMA 通道标志是否设置
ITStatus　DMA_GetITStatus (uint32_t DMA_IT)	检查指定的 DMA 通道中断发生与否
void　DMA_Init (DMA_Channel_TypeDef *DMAy_Channelx，DMA_InitTypeDef *DMA_InitStruct)	初始化通道
void　DMA_ITConfig (DMA_Channel_TypeDef *DMAy_Channelx，uint32_t DMA_IT，FunctionalState NewState)	使能或者禁用指定的 DMA 通道中断
void　DMA_SetCurrDataCounter (DMA_Channel_TypeDef *DMAy_Channelx，uint16_t DataNumber)	设置当前的 DMA 通道传送数据个数
void　DMA_StructInit (DMA_InitTypeDef *DMA_InitStruct)	设置 DMA_InitStruct 成员变量为默认值

（1）DMA 的固件库使用方法。

- 通过 RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1,ENABLE)函数使用 DAM1 控制器时钟使能和配置连接到 DMA 通道的外设（SARM/FLASH 无须初始化）。
- 针对确定通道使用 DMA_Init()函数配置源目标地址、传输方向、缓冲区大小、外设和内存递增模式、数据长度、循环或者正常模式。
- 通过 DMA_ITConfig()函数配置 DMA 中断。
- 通过 DMA_Cmd()函数使能 DMA。除了 SRAM 和 FLASH 外的所有外设均可通过 PPP_DMACmd()函数激活相应通道（PPP 是指相应外设，例如 SPI、USART）。该函数在相应的外设驱动库中，不在 DMA 驱动库中。

（2）为了控制 DMA 事件可以进行的操作。

- 通过 DMA_GetFlagStatus()函数检查 DMA 通道标志。
- 在初始化阶段通过 DMA_ITConfig()函数使能 DMA 中断，在通信阶段可通过 DMA_GetITStatus()函数读取标志。
- 检验完标志后需要通过 DMA_ClearFlag()标志或者 DMA_ClearITPendingBit()清除标志。

12.4　基于 DMA 的 ADC 采样

如果要求 ADC 的多个通道采集数据在每次采集结束后立即读取采集数据，对于不断读取状态方式（即超级循环方式）会面临，因为 ADC 转换数据寄存器只有一个，程序一直等待转换结束标志的尴尬情况。即使采用中断方式也会面临，为了防止丢失转换数据，需要在每个转换结束后触发一次中断进而读 ADC 转换寄存器，中断响应过于频繁的情况。实际上这一切都源于 STM32F0 仅包含一个采样转换寄存器。

为了避免这种情况，可采用 DMA 方式。通过开启 DMA 模式，每次转换结束时都会产

生一个 DMA 请求，从而将 ADC_DR 寄存器中的转换数据传送到软件指定的目标地址，并可利用 DMA 中断在多个通道采集完毕仅进入一次中断，避免中断泛滥现象。

下面代码演示了利用 DMA 进行 ADC 采集的方式。其中 ADC_Config()函数配置 ADC 外设，采用连续转换模式，即转换开始后，一直转换。与非 DMA 模式区别是需要设置 ADC 的 DMA 模式以及使能 DMA 模式。DMA_Config()设置了 DMA 源地址（即 ADC_DR 寄存器地址）以及转换后存取地址（对应程序中&RegularConvData_Tab）。

```
static void ADC_Config(void)
{
  ADC_InitTypeDef       ADC_InitStructure;
  GPIO_InitTypeDef      GPIO_InitStructure;
  ADC_DeInit(ADC1);
  /* GPIOA 外设时钟使能 */
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
   /* ADC1 时钟使能 */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 ;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
  GPIO_Init(GPIOA, &GPIO_InitStructure);
  /* 初始化 ADC*/
  ADC_StructInit(&ADC_InitStructure);
  /* Configure the ADC1 in continuous mode withe a resolution equal to 12 bits  */
  ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
  ADC_InitStructure.ADC_ContinuousConvMode = ENABLE; //连续转换模式
  ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_ScanDirection = ADC_ScanDirection_Backward;
  ADC_Init(ADC1, &ADC_InitStructure);
  ADC_ChannelConfig(ADC1, ADC_Channel_1 , ADC_SampleTime_55_5Cycles);
  /* 校准 */
  ADC_GetCalibrationFactor(ADC1);
  /* ADC DMA 循环方式的 DMA */
  ADC_DMARequestModeConfig(ADC1, ADC_DMAMode_Circular);
  /*使能 ADC_DMA */
  ADC_DMACmd(ADC1, ENABLE);
  /*使能 ADC 外设 */
  ADC_Cmd(ADC1, ENABLE);
  /* 等待 ADRDY 标志 */
  while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_ADRDY));
  /* ADC1 regular Software Start Conv */
  ADC_StartOfConversion(ADC1);
}

/**
  * @brief   DMA channel1 configuration
```

```
  * @param    None
  * @retval None
  */
static void DMA_Config(void)
{
  DMA_InitTypeDef     DMA_InitStructure;
  /* DMA1 时钟设置 */
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1 , ENABLE);
  /* DMA1 通道配置 */
  DMA_DeInit(DMA1_Channel1);
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)ADC1_DR_Address;//外设地址
  DMA_InitStructure.DMA_MemoryBaseAddr    =(uint32_t)&RegularConvData_Tab;    DMA_InitStructure.
DMA_DIR = DMA_DIR_PeripheralSRC;
  DMA_InitStructure.DMA_BufferSize = 1;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
  DMA_Init(DMA1_Channel1, &DMA_InitStructure);
 //设置 DMA 中断
    DMA_ITConfig(DMA1_Channel1,DMA_IT_TC,ENABLE );
  NVIC_InitStructure.NVIC_IRQChannel = DMA1_Channel1_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPriority = 0x01;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
 //
  /* DMA1 通道使能 */
  DMA_Cmd(DMA1_Channel1, ENABLE);



}
```

DMA 传输结束中断函数如下。

```
void DMA1_Channel1_IRQHandler(void)
{
    if (DMA_GetITStatus(DMA1_IT_TC1)!= RESET)
    { endflag=1;
        DMA_ClearITPendingBit(DMA1_IT_TC1);
    }
}
```

本例采用 DMA 中断，每次传输结束后进入中断进行 ADC 采样处理逻辑。如果采用轮询模式，会造成一直读取 DMA 标志位，影响其他任务执行（ST 公司提供的例程为轮询模

式，作为功能演示并无不妥，但用于实际产品开发会造成其他任务无法执行，除非其余任务都是基于采样数据的）。

12.5　小　　结

DMA 对应英文是 Direct Memory Access，简单说就是跳过了 CPU，直接与目标对象交互数据。DMA 这种从外设读取数据的方式是除了通过判断状态位读取数据的轮询方式和中断响应方式之外的另一种高效数据通信方式。DMA 方式是通过 DMA 控制器接管数据线和地址线，无须 CPU 的介入，从而最大限度利用内存宽度和提高 IO 速度。最后本章给出如何通过 DMA 读取 ADC 的采样数据实例。

第13章

串行外设接口/I2S音频（SPI/I2S）

13.1 简　　介

通过软件选择SPI或I2S模式，STM32F0x的SPI/I2S接口可进行SPI或者I2S通信，器件复位默认是SPI模式。

串行外设接口（SPI）协议是采用半双工，全双工和简单同步等方式与外部设备进行串行通信。SPI的通信是以主从方式进行，即一个主设备和一个或者多个从设备（STM32F0xx芯片可充当主设备角色、也可充当从设备角色）。SPI通信协议未规定通信速率，STM32F0xx芯片的速率可达18Mbit/s（对比USART的6Mbit/s，以及I^2C的1Mbit/s）。SPI有四个信号线：时钟信号（SCK）、并行数据线“主机输出，从机输入”（MOSI）、并行数据线“主机输入，从机输出”（MISO）与片选信号。故SPI通信方式可以实现：三线全双工、两线的半双工同步传输，以及两线的简单同步传输方式。

I2S音频协议，也是一个同步串行通信接口，使用3路信号，适用于飞利浦的I2S标准、MSB和LSB对齐的标准、PCM标准，可以实现主从模式的半双工通信。I2S作为主设备的时候可以向外部从设备提供通信时钟。

13.1.1 SPI主要特点

- 主设备或从设备模式。
- 三线全双工同步传输。
- 两条线的半双工同步传输（双向数据线）。
- 两条线的简单同步传输（单向数据线）。
- 4～16位数据的大小选择。
- 多重模式的能力。
- 8个主模式波特率分频器，波特率高达$f_{PCLK}/2$。
- 从模式频率高达$f_{PCLK}/2$。
- 主机和从机模式下都可以由硬件或软件管理NSS：主/从模式操作的动态变化。
- 可编程时钟极性和相位。
- 高位在前或低位在前可设置。
- 专用的发送和接收状态标志，全部支持中断触发。

- SPI 总线忙状态标志。
- SPI 摩托罗拉方式支持。
- 硬件 CRC 功能实现可靠的通信：
  - CRC 值可以作为 TX 模式的最后一个字节传送；
  - 自动 CRC 错误检查上次接收到的字节。
- 主模式故障、溢出等标志具备中断触发能力。
- CRC 错误标志。
- 两个 32 位嵌入式 Rx 和 Tx FIFO 带 DMA 功能。
- 支持 SPI TI 模式。

13.1.2　SPI/I2S 具体功能实现

表 13-1 是 STM32F0x 系列芯片的 SPI 功能实现。

表 13-1　STM32F0x 芯片功能实现

SPI 特征	STM32F03x	STM32F04x STM32F05x		STM32F07x	
	SPI1	SPI1	SPI2	SPI1	SPI2
硬件 CRC 计算	X	X	X	X	X
Rx/Tx FIFO	X	X	X	X	X
NSS+模式	X	X	X	X	X
I2S 模式	X	X		X	X
TI 模式	X	X	X	X	X

13.2　SPI 功能描述

13.2.1　SPI 框图

SPI 允许 MCU 和外部设备之间同步串行通信。应用软件通过状态标志或 SPI 中断来管理通信过程。

注意：图 13-1 相比 STM32F1 系列的框图一个显著区别是，读是通过 RX FIFO，而不是接收缓冲区。SPIx_CR2 寄存器中的 FRXTH 位默认是 0，即对应是 16 位才引起 RXNE，接收 8 位数据时需要设置该位。

图 13-1 是 STM32F0x 的原理框图，通常 SPI 通过 4 个引脚与外部设备相连。

- MISO：主设备输入/从设备输出引脚。一般情况下，此引脚用于在从模式下的数据发送和主模式下的数据接收。
- MOSI：主设备输出/从设备输入引脚。一般情况下，此引脚用于在从模式下的数据接收和主模式下的数据发送。
- SCK：SPI 串行时钟输出引脚，作为主设备输入、从设备的输出。
- NSS：从机片选脚。根据 SPI 和 NSS 设置，此引脚可用于选择一个从机来通信，同步数据帧，检测多个主机之间的冲突。

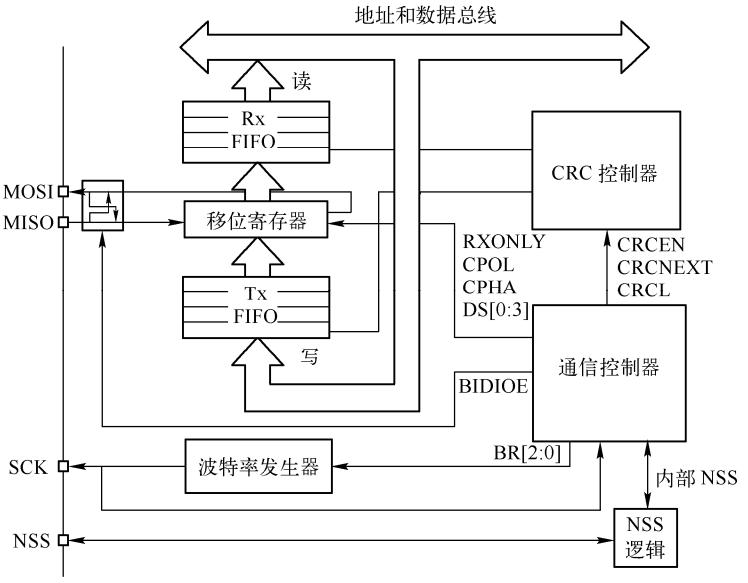


图 13-1　SPI 原理框图

SPI 总线允许一个主设备和一个或多个从设备之间的通信。总线至少有两条线：时钟信号和同步数据传输。其他信号基于 SPI 节点和主机间数据交换目的进行选择。

13.2.2　一主、一从通信

SPI 允许 MCU 根据目标设备和应用程序要求，使用不同的配置进行通信。这些配置包括 2 或 3 线、（软件 NSS 管理）、3 或 4 线（硬件 NSS 管理）。通信总是由主机发起的。通信方式有全双工模式、半双工模式、单发（单收）模式。

1．全双工通信

默认情况下，SPI 被配置为全双工通信，如图 13-2 所示。在此配置中，主机和从机的移位寄存器通过两个单向线（MOSI 和 MISO 引脚）进行连接。在 SPI 通信时，按照主机提供的 SCK 时钟沿进行同步数据传输。主机的数据经由 MOSI 线发送到从机，从机的数据经由 MISO 线发送到主机。当数据帧传输完成（所有位转移）时，主机和从机间的信息交换就完成了。

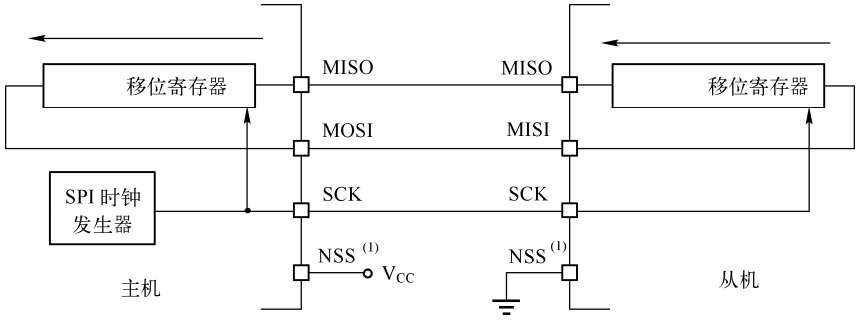


图 13-2　全双工单主/单从应用

注：（1）将 NSS 引脚配置为输入。

2．半双工通信

通过设置 SPIx_CR1 寄存器的 BIDIMODE 位，SPI 工作在半双工（half-duplex）模式下，如图 13-3 所示。在通信期间，数据按照 SPIx_CR1 寄存器的 BDIOE 位设定，由主机移位寄存器同步于 SCK 时钟沿传输到从机。在此配置中，主机的 MISO 引脚和从机的 MOSI 引脚可作为 GPIO 供其他应用程序使用。

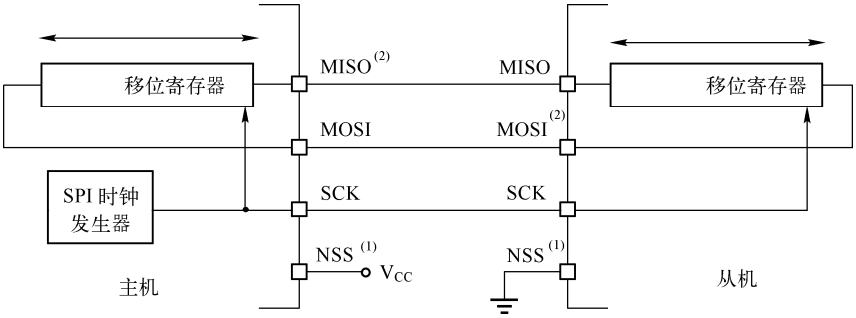


图 13-3　半双工单主/单从应用

注：（1）在这种情况下，NSS 引脚被配置为输入。

（2）在此配置中，主机的 MISO 引脚和从机的 MOSI 引脚可以作为 GPIO 用。

3．单工通信

可以通过 SPIx_CR2 寄存器的 RXONLY 位选择 SPI 工作在单工模式下，实现单发送或单接收，如图 13-4 所示。在此配置中，由一个单一的跨接线连接主机寄存器和从机。其余 MISO 和 MOSI 引脚不用于通信，并可以作为标准的 GPIO 使用。

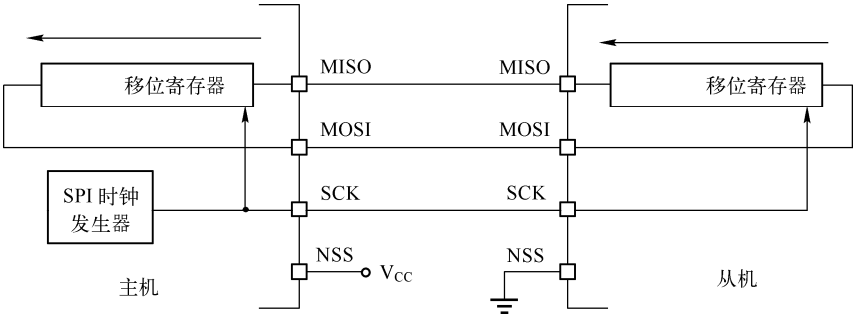


图 13-4　简单的单主/单从应用（主机仅发送/从机仅接收）

单发送模式（RXONLY = 0）：配置设置和全双工相同。应用程序必须忽略在未使用的输入引脚上捕获的信息。这个脚可以作为一个标准的 GPIO 引脚。

单接收模式（RXONLY = 1）：应用程序可以设置 RXONLY 位以禁用 SPI 输出功能。在配置为从机时，MISO 输出被禁用，并且可以当作一个 GPIO 引脚来使用。从机将从 MOSI 引脚连续接收数据，在从机选择信号处于激活状态时，它的 BSY 标志总是为 1（见 13.3.4 节）。根据数据缓冲区的配置，会出现数据接收事件。在配置为主机时，MOSI 输出被禁用，并且可以当作一个 GPIO 引脚来使用。SPI 使能时，时钟信号会不断产生。要停止时钟输出，只有清除 RXONLY 位或 SPE 位，并等待直到从 MISO 引脚的输入样式的完成并填充数据缓冲区结构。这取决于其具体配置。

13.2.3 多从机通信

在有两个或两个以上独立的从机的配置时，主机用 GPIO 引脚来管理每个从机的片选线，如图 13-5 所示。主机必须通过 GPIO 拉低其中一个从机的 NSS 引脚，从而建立一个标准的主机和专用的通信渠道。

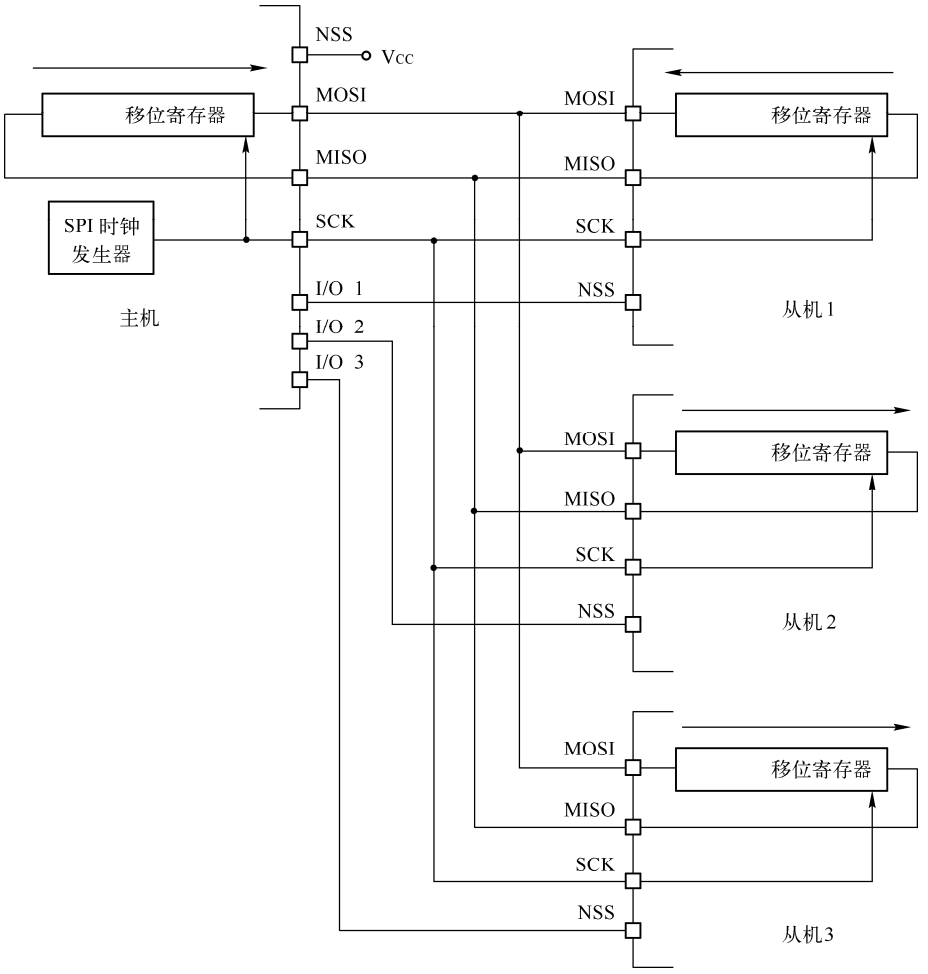


图 13-5 主机和 3 个独立的从机

13.2.4 从机选择（NSS）的引脚管理

在从机模式下时，NSS 作为一个标准的“片选”输入工作，并允许主从通信。在主机模式下，NSS 可以用来作为输入或输出。作为输入，它可以防止多主总线冲突，作为输出，它可以驱动一个单一的从机片选信号。通过 SPIx_CR1 寄存器中的 SSM 位，来选择使用硬件或软件的从机选择管理，如图 13-6 所示。

- 软件 NSS 管理（SSM = 1）：在此配置中，从机选择信息由内部寄存器 SPIx_CR1 的 SSI 位值来驱动。外部 NSS 引脚可以由其他应用程序自由支配。

- 硬件 NSS 管理（SSM = 0）：在这种情况下，有两种可能的配置。这种设置由（寄存器 SPIx_CR1SSOE 位）来决定 NSS 的输出。
  - NSS 输出使能（SSM= 0，SSOE = 1）：此配置仅用于当 MCU 作为主机时。NSS 引脚由硬件管理。在 SPI 作为主模式（SPE 的= 1）的同时 NSS 信号被拉低，并保持低直到 SPI 被禁止（SPE=0）。如果 NSS 脉冲模式被打开（NSSP=1），在连续通信间隔期间可以产生一个 NSS 脉冲。在这种 NSS 设置中，SP 不能支持多重主机工作状态。
  - NSS 输出禁止（SSM = 0，SSOE= 0）：如果在总线上 MCU 作为主机，那么此配置就允许多主机通信。如果在这种模式下 NSS 引脚被拉低，SPI 进入主模式故障状态，并自动重新配置为从机模式。在从机模式下，NSS 引脚作为标准的片选输入来工作，当 NSS 线被拉低时，从机被选中。

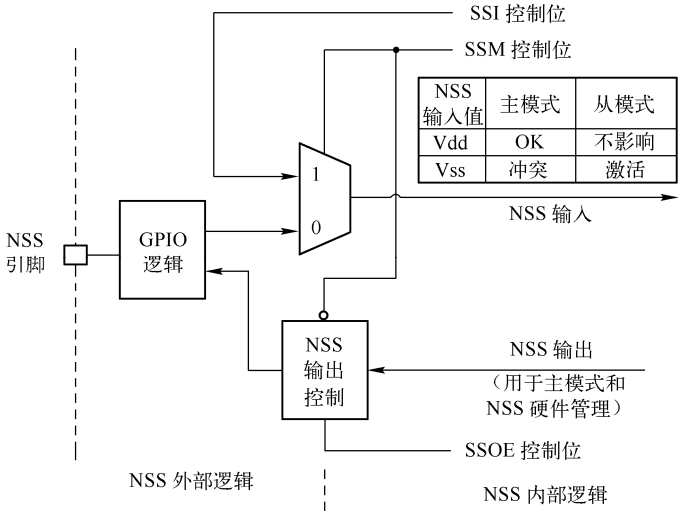


图 13-6　硬件/软件从机选择管理

13.2.5　通信格式

在 SPI 通信中，接收和发送操作同时进行。串行时钟（SCK）同步发送并同时采样数据线上的信息。通信格式取决于时钟相位、时钟极性和数据帧格式。为了能够正常通信，主机和从机必须遵循相同的通信格式。

1．时钟相位和极性控制

可以通过 SPIx_CR1 的 CPOL 和 CPHA 位用软件选择四种可能的时序关系。CPOL（时钟极性）位控制着在没有数据在发送时的空闲状态的时钟输出电平。该位既针对主机模式也针对从机模式。如果 CPOL 被清零，SCK 引脚在闲置状态输出低电平。如果 CPOL 位被置 1，SCK 引脚在闲置状态输出高电平。如果 CPHA 位被置 1，SCK 引脚上的第二沿对准的是第一位的捕获时机（如果 CPOL 位为 0，则为下降沿；如果 CPOL 位为 1，则为上升沿）。每次发生这个时钟切换时，数据被锁存。如果 CPHA 位为 0，SCK 引脚上的第一个沿对准第一位的捕获时机（如果 CPOL 位为 1，则为下降沿；如果 CPO 位为 0，则为上升沿）。每次发

生这个时钟切换时，数据被锁存。CPOL（时钟极性）和 CPHA（时钟相位）的选择共同决定数据采集时的时钟边沿。

2．数据帧格式

SPI 移位寄存器可以设置 MSB 在前或 LSB 在前，取决于 LSBFIRST 位的值。数据字长使用 DS 位选择。它可以设定 4～16 位的长度，同时适用于发送和接收。无论选定多大字长，对 FIFO 的读访问必须与 FRXTH 水平对齐。当访问 SPIx_DR 寄存器时，无论是一个字节（如果数据能够在一个字节中）还是一个字，数据帧总是右对齐。通信时，只有数据字长范围内的位会随时钟输出。

13.2.6 SPI 的初始化

主机和从机的初始化过程几乎是相同的，均通过配置寄存器 SPIx_CR1 和 SPIx_CR2 设置相应位。

① 使用 BR[2:0]位选择串行时钟的波特率。

② 设置 CPOL 和 CPHA 位的组合，定义数据发送和串行时钟之间的四个关系。

③ 配置 RXONLY、BIDIOE 和 BIDIMODE 选择传输模式。

④ 设置 DS 位选择用于传送的数据长度。

⑤ 配置 LSBFIRST 位，定义帧格式。

⑥ 根据应用的需要，设置 SSM 的 SSI 和 SSOE。在主机模式下，内部 NSS 信号必须在完整序列期间保持为高。在从机模式下，内部 NSS 信号必须在完整序列期间保持为低。

⑦ 如果需要按 TI 协议，设置 FRF 位。

⑧ 如果两个数据单位之间需要 NSS 脉冲，那么设置 NSSP 位以打开 NSS 脉冲模式。此配置下 CHPA 位必须设置为 1。

⑨ 设置 FRXTH 位。RXFIFO 的阈值必须和对 SPIx_DR 寄存器的读访问的字长对齐。

⑩ 如果使用 DMA，初始化 LDMA_TX 和 LDMA_RX 位。

⑪ 如果需要 CRC，将 CRC 多项式设置为“输入”并设置 CRCEN 位。

⑫ 在 NSS 内部信号为高的时候设置 MSTR 位。

⑬ 设置 SPE 位，启用 SPI。

13.2.7 数据发送和接收流程

1．RXFIFO 和 TXFIFO

所有 SPI 数据交换都通过 32 位的嵌入式 FIFO。这使 SPI 可以连续工作，防止短数据帧时的数据断流。每个方向都有它自己的 FIFO，称为 TXFIFO 和 RXFIFO。这些 FIFO 被用于除了 CRC 使能的单接收模式外的所有的 SPI 模式（主从）。

对 FIFO 的处理会根据数据交换模式（双工，单工）、数据帧格式（字长）、对 FIFO 数据寄存器访问的大小（8 位或 16 位），以及是否按照数据包访问 FIFO。

对 SPIx_DR 寄存器的读访问，会返回存储在 RXFIFO 中但还未读的最近的数据。对 SPIx_DR 的写访问会将新的数据放在发送队列的尾部。读访问必须总是和 SPIx_CR2 寄存器中的 FRXTH 位设置的 RXFIFO 门限对齐。FTVL[1:0]和 FRLV[1:0]位表明两个 FIFO 目前的缓冲存储程度。

对 SPIx_DR 寄存器的读访问必须由 RXNE 事件引发。这个时间在数据被存入 RXFIFO 并达到门槛（由 FRXTH 位定义）的时候被触发。在 RXNE 被清除时，认为是空的 RXFIFO 已经清空。同样写访问要由 TXE 事件来触发。当 TXFIFO 的存储状况少于或等于容量一半时，将触发这个事件。否则 TXE 被清零，并且 TXFIFO 被认为是有数据。用这种方式，RXFIFO 可以存储多达 4 个数据帧，而 XFIFO 在字长不大于 8 的时候也只可以存储多达 3 个数据帧。这种差异可以防止在已经有 3 个 8 位数据存在 TXFIFO 中的时候，软件试图在 16 位模式下向 TXFIFO 写入更多的数据而造成数据破坏。TXE 和 RXNE 事件都可以通过中断查询或处理。

如果接收到下一个数据时 RXFIFO 是满的，将导致发生溢出事件。溢出事件可以查询处理或中断处理。正在执行数据传输的时候，硬件会将 BSY 位置 1 来指示这个状态。当时钟信号连续运行时，如果是主机模式，BSY 标志在帧与帧之间一直保持为 1，而在从机模式时，BSY 信号在帧与帧之间会有一小段时间（1 个 SPI 时钟周期）为 0。

2．序列处理

多个数据字节可以顺序发送来组成一个消息。启用发送后，在主机 TXFIFO 中的数据会开始发送并连续发完。主机会连续输出时钟信号，直至 TXFIFO 变为空，然后停下来等待新增的数据。

在单接收模式、半双工（BIDIMODE=1，BIDIOE=0）模式或简单发送（BIDIMODE=0，RXONLY=1）模式下，只要 SPI 和单接收模式被使能，主机会立即开始接收序列。主机连续提供时钟信号，直到主机关闭 SPI 或者关闭单接收模式之前都不会停下来。这时主机会连续接收数据。

数据帧开始后，从机无法控制或延迟数据序列。出于这个原因，从机必须在开始传输之前准备好数据，并总是一直保持有数据待传（存在 TXFIFO 中）。主机必须在每个序列之间给从机保留足够的时间以准备数据。如果可能的话，序列中的字节的数量应得到限制，以便从机完成自动的数据处理（通过使用 DMA 或 FIFO）。主机必须提供额外的时间给从机处理数据内容。

在多从机并行的系统中，每个序列应该由 NSS 脉冲来分隔，以将每个序列对应到不同的从机。在单从机系统中通过 NSS 控制从机就显得没那么有必要了，但这里有个脉冲还是更好，这可以令从机和每个数据序列完成同步。NSS 既可以由软件管理也可以由硬件管理。

BSY 位被置 1 时，它标志着一个持续的传输。结合 FTLV[1:0]位，用于检查传输是否完成。传输完成，RXNE 标志被置 1，即最后一位刚被采集，完整的数据帧存储在 RXFIFO 中。

3．数据打包

当数据帧的大小适合一个字节（小于或等于 8 位），并且对 SPIx_DR 寄存器执行任何 16 位的读/写访问时，数据会自动打包在一起。在这种情况下，可以并行处理双数据。SPI 先操作低 8 位，然后操作高 8 位。在一次对 SPIx_DR 的 16 位写访问后，就会有 2 个字节的数据被发送出去。如果 RXFIFO 的阈值设置为 16 位（FRXTH=0），该序列则只会生成一个接收 RXNE 事件，而不是两个。针对这种单个的 RXNE 事件的响应，接收器必须对 SPIx_DR 寄存器作一次 16 位的读访问才能够把数据全都取到。RXFIFO 的阈值和跟进的数据访问的位

宽必须保持一致，否则就会丢数据。

如果出现奇数个字节数据，那就会出现特别的问题，这是一定要解决的。在发送端，用 8 位方式访问 SPIx_DR 将最后一个字节发出来就够了。在接收端必须改变 Rx_FIFO 门限，以便在传送奇数字节的数据帧的最后字节时能够产生 RXNE 事件。

13.2.8 状态标志

TX 缓冲器空标志（TXE）、Rx 缓冲非空（RXNE）、忙标志（BSY 变）三个标志位限制了 SPI 总线的状态。

当 TXFIFO 中有了足够的空间来存储发送数据时，TXE 标志被置 1。TXE 标志是连到 TXFIFO Level 的。在 TXFIFO 的存储水平低于或等于整个 FIFO 深度的一半时，这个标志会置高并且保持高。如果 SPIx_CR2 中的 TXEIE 位是 1，还会产生一个中断。如果 TXFIFO 的存储水平又高于 FIFO 深度的一半了，那这个位会自动地清零。

RXNE 标志的设置取决于 SPIx_CR2 寄存器的 FRXTH 位值：

- 如果 FRXTH 为 1，RXNE 会在 RXFIFO 的存储水平大于或等于 1/4（8 位）的时候被置高，并且保持住。
- 如果 FRXTH 为 0，RXNE 会在 RXFIFO 的存储水平大于或等于 1/2（16 位）的时候被置高，并且保持住。

如果这时 SPIx_CR2 寄存器中的 RXNEIE 位是 1，那就会产生一个中断请求。当上述条件不再成立时 RXNE 由硬件自动清零。

BSY 标志由硬件设置和清零（软件改写这个标志是没有用的）。当 BSY 为 1 时，它表示 SPI 正处于数据传输过程中（SPI 总线忙）。在某些模式下可以使用 BSY 标志来检测传输结束，从而软件可以在进入 HALT 模式之前禁用 SPI 及其外设时钟。这就避免了破坏最后的传输字节。BSY 标志在防止多主机系统中的写碰撞也是很有用的。BSY 标志在下列条件之一达成时被清除。

- 当 SPI 被正确禁用时；
- 当在主模式（MODF 位设置为 1）中检测到故障时；
- 在主模式下，当完成了数据发送，并且没有新的数据要发送时；
- 在从模式下，当两次数据传输之间间隔达到至少一个 SPI 时钟周期时。

13.2.9 错误标志

错误标志包含溢出标志（OVR）、模式故障位（MODF）、CRC 错误（CRCERR）、TI 模式帧格式错误（FRE）。在将错误中断使能位 ERRIE 置 1 后，如果下列错误标志被置 1，就会产生 SPI 中断。

当主机或从机在收到数据之后不去清除 RXNE 位，然后收到了新的数据之后，会引发溢出错误（OVR）。当接收数据时 RXFIFO 中没有足够的空间存储收到的数据时，溢出的情况也会发生。如果软件或者 DMA 没有足够时间去读取 RXFIFO 中已收到数据，来为后面的数据腾出足够的空间时就会发生这种事。在 CRC 功能打开时，接收缓冲区会被认为是一个单缓冲区，这导致 RXFIFO 不好使了，这也会直接导致溢出事件。

溢出的情况发生时，新收到的值不会覆盖在 RXFIFO 的前一个数据。新收到的值将被丢

弃，随后传输的所有数据都将丢失。在读了 SPIx_SR 寄存器后，再去读一下 SPIx_DR 寄存器，就会清除 OVR 位。

主机得知其内部 NSS 信号（NSS 引脚为硬件模式，或在 NSS 软件模式 SSI 位）被拉低时，发生模式故障位。这将自动设置 MODF 位。主机模式故障对 SPI 接口的影响包括以下方面：

- MODF 位被置 1。另外如果 ERRIE 位被置 1，会产生 SPI 中断。
- SPE 位被清零。这将阻止主机的数据输出，同时禁用 SPI 接口。
- MSTR 位被清除，从而迫使设备转到从机模式。

使用下面的软件序列，以清除 MODF 位：

- 在 MODF 位被设置后，对 SPIx_SR 寄存器做一次读或写访问。
- 然后写一下 SPIx_CR1 寄存器。

为了避免系统中的多个从机发生冲突，NSS 引脚必须在 MODF 位清零过程中拉高。SPE 和 MSTR 位可以在这个清零过程后恢复到原来的状态。作为一种安全措施，硬件不允许在 MODF 位为 1 期间将 SPE 的 MSTR 位置 1。在从模式下，MODF 位永远不可能被置 1，除非是以前的多主机冲突的结果。

在 SPIx_CR1 寄存器中的 CRCEN 位为 1 时，这个标志用来确认收到的数据的有效性。如果接收移位寄存器中的值和接收 SPIx_RXCRCR 值不匹配，SPIx_SR 寄存器中的 CRCERR 标志会被置 1。该标志由软件清除。

当 SPI 工作在从模式、TI 模式协议时，如果数据通信期间，在 NSS 上出现一个脉冲时，会引起一次 TI 模式帧格式错误。发生此错误时，SPIx_SR 寄存器中的 FRE 的标志会被置 1。在发生错误时 SPI 不会被禁止，NSS 脉冲会被忽略，SPI 会在开始新的传输前等待下一个 NSS 脉冲。因为错误检测可能会导致两个字节的数据丢失，数据可能会损坏。读 SPIx_SR 寄存器，FRE 的标志会被清除。如果 ERRIE 位为 1，会在 NSS 错误发生后产生一个中断。在这种情况下，SPI 应该被禁用，因为数据的完整性将不能保证了，通信应该重新开始。先重新使能从机，再重新初始化主机就行了。

13.3　SPI 中断

表 13-2 是 SPI 的中断请求。

表 13-2　SPI 中断请求

中断事件	事件标志	使能控制位
发送 TXFIFO 待装填	TXE	TXEIE
在接收 RXFIFO 中有收到的数据	RXNE	RXNEIE
主模式故障	MODF	ERRIE
溢出错误	OVR	
CRC 错误	CRCERR	
TI 帧格式错误	FRE	

13.4 SPI 固件库

在 stm32f0xx_spi.h 与 stm32f0xx_spi.c 包含了 SPI 的固件库。表 13-3 是 SPI 的固件库函数。

表 13-3 固件库函数

组	函 数	描 述
初始化和配置函数	void SPI_I2S_DeInit(SPI_TypeDef* SPIx)	恢复寄存器值为默认值
	void SPI_Init(SPI_TypeDef* SPIx, SPI_InitTypeDef* SPI_InitStruct)	使用输入参数初始化 SPI
	void I2S_Init(SPI_TypeDef* SPIx, I2S_InitTypeDef* I2S_InitStruct), STM32F030 devices 不可用	使用默认参数初始化 I2S
	void SPI_StructInit(SPI_InitTypeDef* SPI_InitStruct)	SPI_InitStruct 结构体初始化默认值
	void I2S_StructInit(I2S_InitTypeDef* I2S_InitStruct), STM32F030 不可用	设置 I2S_InitStruct 默认值
	void SPI_TIModeCmd(SPI_TypeDef* SPIx, FunctionalState NewState)	使能或禁用 TI 模式
	void SPI_NSSPulseModeCmd(SPI_TypeDef* SPIx, FunctionalState NewState)	使能或禁用 SPI 的 NSS 模式
	void SPI_Cmd(SPI_TypeDef* SPIx, FunctionalState NewState)	使能或禁用 SPI 外设
	void I2S_Cmd(SPI_TypeDef* SPIx, FunctionalState NewState), STM32F030 不可用	使能或禁用 SPI 外设 I2S 模式
	void SPI_DataSizeConfig(SPI_TypeDef* SPIx, uint16_t SPI_DataSize)	配置 SPI 的数据长度
	void SPI_RxFIFOThresholdConfig(SPI_TypeDef* SPIx, uint16_t SPI_RxFIFOThreshold)	配置 SPI 接收门限
	void SPI_BiDirectionalLineConfig(SPI_TypeDef* SPIx, uint16_t SPI_Direction)	选用 SPI 双向模式中的传送方向
	void SPI_NSSInternalSoftwareConfig(SPI_TypeDef* SPIx, uint16_t SPI_NSSInternalSoft)	指定 SPI 的 NSS 内部状态
	void SPI_SSOutputCmd(SPI_TypeDef* SPIx, FunctionalState NewState)	使能或禁用 SS 输出
数据传递函数	void SPI_SendData8(SPI_TypeDef* SPIx, uint8_t Data)	发送数据
	void SPI_I2S_SendData16(SPI_TypeDef* SPIx, uint16_t Data)	发送数据
	uint8_t SPI_ReceiveData8(SPI_TypeDef* SPIx)	接收数据
	uint16_t SPI_I2S_ReceiveData16(SPI_TypeDef* SPIx)	接收数据
硬件 CRC 计算	void SPI_CRCLengthConfig(SPI_TypeDef* SPIx, uint16_t SPI_CRCLength)	配置 CRC 计算长度
	void SPI_CalculateCRC(SPI_TypeDef* SPIx, FunctionalState NewState)	CRC 计算
	void SPI_TransmitCRC(SPI_TypeDef* SPIx)	发送 CRC 值
	uint16_t SPI_GetCRC(SPI_TypeDef* SPIx, uint8_t SPI_CRC)	读取发送或接收到的 CRC 值
	uint16_t SPI_GetCRCPolynomial(SPI_TypeDef* SPIx)	读取 CRC 的多项式
DMA 传输函数	void SPI_I2S_DMACmd(SPI_TypeDef* SPIx, uint16_t SPI_I2S_DMAReq, FunctionalState NewState)	使能或禁用 DMA 接口
	void SPI_LastDMATransferCmd(SPI_TypeDef* SPIx, uint16_t SPI_LastDMATransfer)	设置发送和接收 DMA 使能
中断和标志设置函数	void SPI_I2S_ITConfig(SPI_TypeDef* SPIx, uint8_t SPI_I2S_IT, FunctionalState NewState)	配置中断
	uint16_t SPI_GetTransmissionFIFOStatus(SPI_TypeDef* SPIx)	读取发送 FIFO 状态
	uint16_t SPI_GetReceptionFIFOStatus(SPI_TypeDef* SPIx)	读取接收 FIFO 状态
	FlagStatus SPI_I2S_GetFlagStatus(SPI_TypeDef* SPIx, uint16_t SPI_I2S_FLAG)	读取标志状态
	void SPI_I2S_ClearFlag(SPI_TypeDef* SPIx, uint16_t SPI_I2S_FLAG)	清除状态标志
	ITStatus SPI_I2S_GetITStatus(SPI_TypeDef* SPIx, uint8_t SPI_I2S_IT)	读取 SPI 中断状态

13.5　SPI 相互通信实例

通过 SPI 外设，可实现两个 STM32F0x 芯片的数据通信，主要用于两个 MCU 共存于一个产品的情况，且 MCU 之间需交互数据。首先将两个 STM32F0x（经测试，STM32F0x 系列芯片均可）的 PA0.5、PA0.6、PA0.7 相互直连，不需接 NSS 引脚（见图 13-2）。

下面以主机通信为主描述代码实现，备机有差别地方加以说明。将主机与从机配置相同的部分统一放在 SPI_Config 函数中。由于是同类型的芯片通信，所以时钟以及时钟极性都设置相同。如果是不同厂家的芯片通信除了保证通信速率相同，还要注意时钟相位和极性以及数据帧格式（见 13.2.5 节）。SPI 通信配置成双线全双工模式，如图 13-2 所示。

```
static void SPI_Config(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;

  /*  使能 SPI 外设  */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);

  /*使能 SCK、MOSI、MISO 的 GPIO 时钟  */
  RCC_AHBPeriphClockCmd(SPIx_SCK_GPIO_CLK|SPIx_MISO_GPIO_CLK|SPIx_MOSI_GPIO_
CLK, ENABLE);
  GPIO_PinAFConfig(SPIx_SCK_GPIO_PORT, SPIx_SCK_SOURCE, SPIx_SCK_AF);
  GPIO_PinAFConfig(SPIx_MOSI_GPIO_PORT, SPIx_MOSI_SOURCE, SPIx_MOSI_AF);
  GPIO_PinAFConfig(SPIx_MISO_GPIO_PORT, SPIx_MISO_SOURCE, SPIx_MISO_AF);

  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd   = GPIO_PuPd_DOWN;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_Level_3;

  /* SPI SCK 引脚配置  */
  GPIO_InitStructure.GPIO_Pin = SPIx_SCK_PIN;
  GPIO_Init(SPIx_SCK_GPIO_PORT, &GPIO_InitStructure);

  /* SPI MOSI 引脚配置  */
  GPIO_InitStructure.GPIO_Pin =   SPIx_MOSI_PIN;
  GPIO_Init(SPIx_MOSI_GPIO_PORT, &GPIO_InitStructure);

  /* SPI MISO 引脚配置  */
  GPIO_InitStructure.GPIO_Pin = SPIx_MISO_PIN;
  GPIO_Init(SPIx_MISO_GPIO_PORT, &GPIO_InitStructure);

  /* SPI  配置*/
```

```
    SPI_I2S_DeInit(SPI1);
    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    SPI_InitStructure.SPI_DataSize = SPI_DATASIZE;
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStructure.SPI_CRCPolynomial = 7;

    /* 配置 SPI 中断优先级 */
    NVIC_InitStructure.NVIC_IRQChannel = SPI1_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

下面代码是初始化 SPI、使能 FIFO、使能 SPI 中断、使能 SPI。熟悉 STM32F1X 芯片的读者需要注意 SPI_RxFIFOThresholdConfig(SPI1, SPI_RxFIFOThreshold_QF)这一句，STM32F0x 的 SPI 是通过 FIFO 通信的。从机的 SPI 设置需将 SPI 模式设置成 SPI_Mode_Slave。其余主机与从机配置相同。

```
SPI_Config();
  /* 初始化 SPI 配置 */
  SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
  SPI_Init(SPI1, &SPI_InitStructure);

  /* 使能 FIFO 阈值 */
  SPI_RxFIFOThresholdConfig(SPI1, SPI_RxFIFOThreshold_QF);
  /* 使能接收中断 */
  SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_RXNE, ENABLE);
  /* 使能 SPI 错误中断 */
  SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_ERR, ENABLE);

  /*使能 SPI 外设 */
  SPI_Cmd(SPI1, ENABLE);
```

SPI 发射代码如下，通过发送缓冲区空标志，使能中断发送。通过判断 FIFO 空以及忙标志判断是否发送完毕（见 13.2.7 节）。

```
    /* 使能发送缓冲区空标志，即开始发送*/
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_TXE, ENABLE);

    /* 等待 TX FIFO 空 */
    while (SPI_GetTransmissionFIFOStatus(SPI1) != SPI_TransmissionFIFOStatus_Empty)
    {}
    /* 等待忙标志 */
```

```
    while(SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_BSY) == SET)
    {}
```

在 SPI 中断中发送数据以及接收数据。主机和从机的中断程序完全相同。

```
void SPI1_IRQHandler(void)
{ //发送
  if (SPI_I2S_GetITStatus(SPI1, SPI_I2S_IT_TXE) == SET)
  {
      SPI_SendData8(SPI1, TxBuffer[Tx_Idx++]);
      if (Tx_Idx == GetVar_NbrOfData())//发送完毕，禁用发送中断
      {
        SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_TXE, DISABLE);
      }
  }
   //接收
  if (SPI_I2S_GetITStatus(SPI1, SPI_I2S_IT_RXNE) == SET)
  {    RxBuffer[Rx_Idx++] = SPI_ReceiveData8(SPI1);
  }
  /* SPI 错误中断 */
  if (SPI_I2S_GetITStatus(SPI1, SPI_I2S_IT_OVR) == SET)
  {
    SPI_ReceiveData8(SPI1);
    SPI_I2S_GetITStatus(SPI1, SPI_I2S_IT_OVR);
  }
}
```

虽然该程序实现了相互通信，但在产品设计时为了保证正常的数据传输，需要配合相应通信协议，即双方通信数据格式和内容约定（可参照一些 SPI 接口的 EEPROM，定义读/写指令、地址、数据等内容）。另外 SPI 通信需注意通信距离受限以及干扰影响，最好是局限于板载情况，否则考虑其他通信方式例如 RS-232、CAN 等。

13.6　小　　结

本章首先讲解了 SPI 通信原理，同时说明了 STM32F1X 系列 SPI 差别。然后对 SPI 通信方式（一主一从、多从方式）和全双工、半双工、单工模式通信，以及通信格式和状态位、中断等内容进行了详细讲解。最后给出了通过 SPI 进行 STM32F0x 芯片间通信的实例，说明了该实例的注意事项。

第14章

I²C 接口

I²C（芯片间)总线接口是连接微控制器和串行 I²C 外设的总线。STM32F0 的 I²C 接口控制所有 I²C 总线特定的时序、协议、仲裁和定时，支持标准模式、快速模式和超快速模式，与 SMBus（系统管理总线）和 PMBus（电源管理总线）保持兼容，可以使用 DMA 减轻 CPU 的负担。

14.1 I²C 的主要特点

除了 SPI、USART 以外，I²C 是的第三种常用板载形式的总线，是一种“线或”形式的总线，即两信号线默认为高，除非被连接设备拉低。图 14-1 中的两个电阻为上拉电阻，使得两信号线保持高电平。图 14-1 中是一种双向两线总线，一条为串行数据线（SDA），一条是串行时钟线（SCL）。由主机通过 SCL 发出时钟信号，主机与从机在 SDA 上发送数据。STM32F0x 支持主机、从机模式。

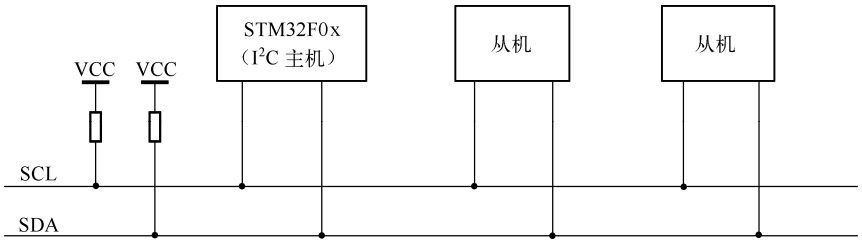


图 14-1　典型 I²C 配置

STM32F0x 的 I²C 有三种形式的操作方式：读、写或者混合方式。有两种形式的地址模式：7 位模式和 10 位模式，其中 10 位地址模式可支持较多设。

相比 STM32F0 的 SPI 的 18Mbps 的通信速率，I²C 的速率就显得稍微慢了点，I²C 的速率：标准模式下可达 100kbit/s，快速模式下可达 400kbit/s，高速模式下可达 3.4Mbit/s。STM32F0x 主要支持标准模式（最高 100kbit/s）和快速模式（最高 400kbit/s）。

表 14-1 是 STM32F0x 家族的 I²C 性能情况。

表 14-1　STM32F0x 的 I²C 性能对比

I2C 特征	STM32F03x STM32F04x	STM32F05x		STM32F07x	
	I²C1	I²C1	I²C2	I²C1	I²C2
独立时钟	X	X		X	
SMBus	X	X		X	
从停止模式唤醒	X	X		X	
FM+模式下 20mA 输出驱动	X	X		X	X

14.2　I²C 功能描述

14.2.1　I²C1 框图

I²C1 的原理框图如 14-2 所示，I²C1 由一个独立的时钟源驱动，可以是 HIS（高速内部振荡器）、SYSCLK（系统时钟）。I²C 时钟与 PCLK 无关联。STM32F03X 系列相比 STM32F0x 家族中的其他成员不包含 I²C 唤醒功能，故原理图中无此模块部分。

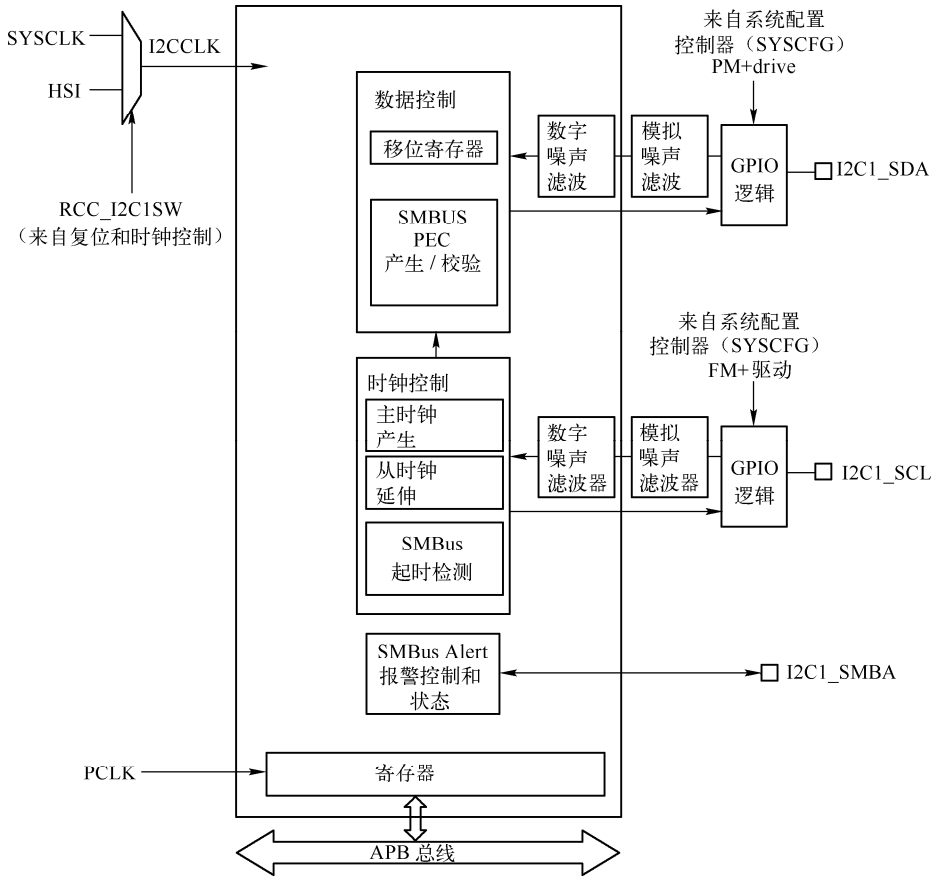


图 14-2　I²C1 原理框图

I^2C 1 的 I/Os 支持 20mA 的输出电流驱动以适应超快速模式的操作。通过将 SCL 和 SDA 的驱动能力控制位置 1 来启用此设置。

14.2.2 I^2C 模式

I^2C 有以下四个模式：

- 从机发送：串行数据通过 SDA 传输，而串行时钟通过 SCL 输入。
- 从机接收：串行数据和串行时钟通过 SDA 和 SCL 接收。
- 主机发送：串行数据通过 SDA 输出，而 SCL 输出串行时钟。
- 主机接收：串行数据通过 SDA 接收，而 SCL 输出串行时钟。

默认情况下，是从机模式。I^2C 接口在生成起始条件后自动地从从模式切换到主模式；当仲裁丢失或产生停止信号时，则从主模式切换到从模式。允许多主机功能。

在主控模式下，I^2C 接口启动数据传输，并产生时钟信号。串行数据传输总是以起始条件开始并以停止条件结束。起始条件和停止条件都是在主模式下由软件控制产生的。起始条件是指时钟线保持高电平期间数据线电平从高到低的跳变作为 I^2C 总线的起始条件。停止条件是指时钟线保持高电平期间数据线电平从低到高的跳变作为 I^2C 总线的停止条件。

从模式时，I^2C 接口能识别它自己的地址（7 位或 10 位）和广播呼叫地址。软件能够控制开启或禁止广播呼叫地址的识别。保留的 SMBus 地址，也可以由软件启用。

数据和地址按 8 位/字节进行传输，高位在前。跟在起始条件后的 1 或 2 个字节是地址（7 位模式为 1 个字节，10 位模式为 2 个字节）。地址只在主模式发送。

在一个字节传输的 8 个时钟后的第 9 个时钟期间，接收器必须回送一个应答位（ACK）给发送器。

图 14-3 所示的协议表明：

① 只有在总线空闲时才允许启动数据传送。

② 在数据传送过程中，当时钟线为高电平时数据线必须保持稳定状态不允许有跳变。时钟线为高电平时数据线的任何电平变化将被看作总线的起始或停止信号。

软件可以开启或禁止应答（ACK），并可以设置 I^2C 接口的地址（7 位、10 位地址或广播呼叫地址）。

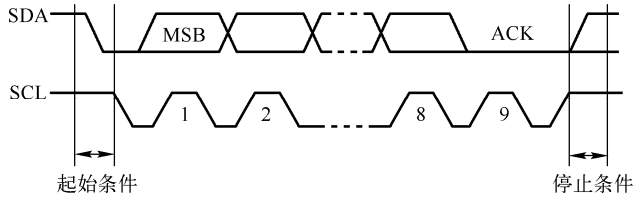


图 14-3 I^2C 总线协议

14.2.3 I^2C 的初始化

I^2C 初始化只要启用外设、配置噪音滤波器和配置 I^2C 的时序。图 14-4 是 I^2C 的初始流程图。

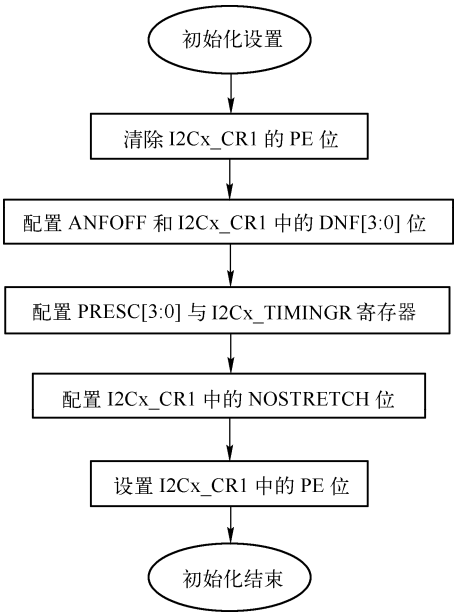


图 14-4　I²C 初始化流程图

1．启用和禁用外设

I²C 外设时钟必须在时钟控制器中配置并启用。然后，I²C 可以通过设置在 I2Cx_CR1 寄存器的 PE 位来使能。当 I²C 被禁用（PE=0）的时候，I²C 执行软件复位：I²C 线（SDA 和 SCL）都被释放，内部状态机复位，所有的通信控制位和状态位回到它们的复位值。

2．噪声滤波器

如有必要，在通过设置在 I2Cx_CR1 寄存器的 PE 位，启用 I²C 外设之前，必须先配置噪声滤波器。默认情况下，模拟噪声滤波器会处理 SDA 和 SCL 输入。这个模拟滤波器符合快速模式和快速模式 Plus I²C 规范的需要，来抑制 50ns 以内的尖峰脉冲宽度。可以通过设置的 ANFOFF 位来禁用这个模拟滤波器和/或配置 I2Cx_CR1 寄存器的 DNF[3:0]位选择数字滤波器。

警告：当 I²C 启用时不允许更改过滤器的配置。

3．I²C 的时序

在主从模式中必须配置时序，以保证正确的数据保持和建立时间。这通过编程 I2Cx_TIMINGR 寄存器中的 PRESC[3:0]、SCLDEL[3:0]和 SDADEL[3:0]位来实现。

14.2.4　数据收发

数据的收发是通过发送和接收数据寄存器和一个移位寄存器进行的。

图 14-5 是接收过程。SDA 输入会填充移位寄存器。在第 8 个 SCL 脉冲（当收到完整的数据字节）之后，如果 I2Cx_RXDR 寄存器是空的（RXNE=0），移位寄存器的内容会被复制到 I2Cx_RXDR 寄存器。如果 RXNE=1，也就是说，尚未读取之前收到的数据字节，这时 SCL 线被拉低直到 I2Cx_RXDR 被读取。这个拉低被插入到第 8 和第 9 个 SCL 脉冲之间（应

答脉冲之前）。

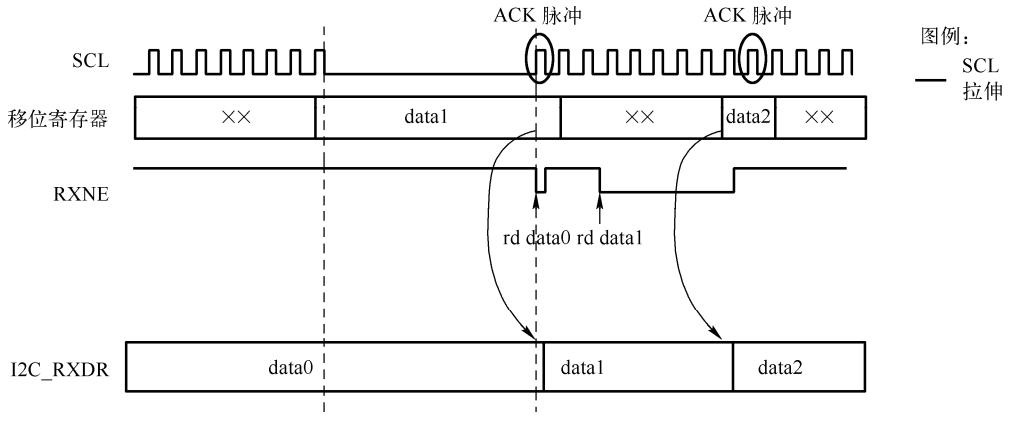


图 14-5　数据接收

图 14-6 是发送过程。如果 I2Cx_TXDR 寄存器不为空（TXE=0），其内容将在第 9 个 SCL 脉冲（应答脉冲）之后被复制到移位寄存器，然后移位寄存器的内容被移到 SDA 线上。如果 TXE=1，也就是说，I2Cx_TXDR 中没有数据被写入，SCL 线会被拉低直到 I2Cx_TXDR 有新数据被写入。在第 9 个 SCL 脉冲之后完成拉低。

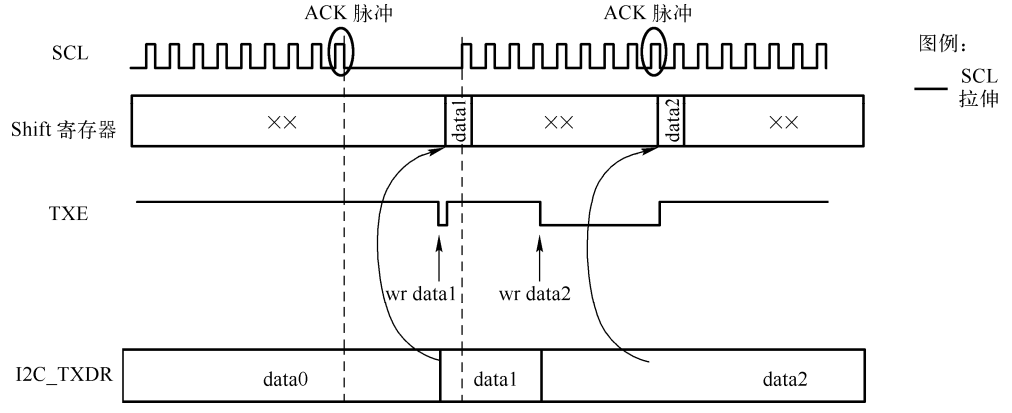


图 14-6　数据发送

I^2C 有一个内嵌的字节计数器，用于管理发送的字节数，以便在各种模式中关闭通信。

- 主模式下生成 NACK、STOP 和 ReSTART；
- 从机接收模式下的 ACK 控制；
- SMBus 的功能支持时的 PEC 的生成/检查。

字节计数器总是在主模式下使用。默认情况下，它在从模式下被禁用，但它可以通过软件设置 I2Cx_CR2 寄存器的 SBC 位（从字节控制）来启用。

要传输的字节数编程在 I2Cx_CR2 寄存器的 NBYTES[7:0]位域。如果要传输的字节数（nBytes 个）大于 255，或一个接收器要控制收到的数据字节的应答值，就必须设置 I2Cx_CR2 寄存器的 RELOAD 位来选择重载模式。在这种模式下，当 NBYTES 中编程的字

节个数被发送完后，TCR 标志被置 1，如果 TCIE 为 1，则会产生一个中断。SCL 在 TCR 标志为 1 期间被拉伸。向 NBYTES 写入一个非零的值时，TCR 标志由软件清除。

当 NBYTES 计数器被重载为上次的字节数时，RELOAD 位必须被清零。当主模式下 RELOAD=0，计数器可以按下列两种模式工作。

- 自动结束模式（I2Cx_CR2 寄存器的 AUTOEND=1）：一旦发送字节数达到了 NBYTES[7:0]位域中设置的字节数，主机会自动发送一个停止条件。
- 软件结束模式（I2Cx_CR2 寄存器的 AUTOEND=0）：发送字节数达到了 NBYTES[7:0]位域中设置的字节数后需要软件的干预，这时 TC 标志会被置 1，如果 TCIE 位为 1，还会产生中断。在 TC 标志为 1 期间，SCL 信号被拉伸。在 I2Cx_CR2 寄存器的 START 或 STOP 位被置 1 时，TC 标志由软件清除。主机要发送一个 RESTART 条件时，必须使用此模式。

注意：AUTOEND 位在 RELOAD 被置 1 时没有效果。

14.2.5　I²C 从机模式

1．I²C 从机初始化

从机模式须启用至少一个从机地址。两个寄存器 I2Cx_OAR1 和 I2Cx_OAR2 都可以用来写入从机的本机地址 OA1 和 OA2。

- 通过设置 I2Cx_OAR1 寄存器的 OA1MODE 位，可将 OA1 配置在 7 位模式（默认）或 10 位地址模式。通过设置 I2Cx_OAR1 寄存器的 OA1EN 位来启用 OA1。
- 如果需要额外的从机地址，可配置第二个从机地址 OA2。配置 I2Cx_OAR2 寄存器的 OA2MSK[2:0]位域，可以最多屏蔽 OA2 的低 7 位。因此从 1 到 6 为 OA2MSK 配置，只有 OA2[7:2]、OA2[7:3]、OA2[7:4]、OA2[7:5]、OA2[7:6]或 OA2[7]参加与接收到的地址的比较操作。只要 OA2MSK 不等于 0，被 OA2 地址比较排除的 I²C 地址（0000xxx 和 1111xxx）不会被应答。如果 OA2MSK=7，收到的所有 7 位地址（保留地址除外）都会被应答。OA2 始终是一个 7 位地址。如果它们启用特定的使能位，这些保留地址也可以应答，就是如果它们被写在 I2Cx_OAR1 或 I2Cx_OAR2 寄存器中并且 OA2MSK=0。通过设置 I2Cx_OAR2 寄存器的 OA2EN 位来启用 OA2。
- I2Cx_CR1 寄存器的 GCEN 位被置 1 时，呼叫地址被启用。

当 I²C 由启用了的地址选中时，ADDR 中断状态标志被置 1，如果 ADDRIE 位为 1，就会产生一个中断。

默认情况下，从机使用它的时钟延长的功能，这意味着，它可根据需要将 SCL 信号拉低以便执行软件动作。如果主机不支持时钟延长，I²C 的 I2Cx_CR1 寄存器的 NOSTRETCH 必须配置为 1。

收到地址中断后，如果启用了多个地址，必须读 I2Cx_ISR 寄存器中的 ADDCODE[6:0]位以检查是哪一个地址匹配上了。还要检查 DIR 标志，以了解数据传输的方向。

在默认模式下，I²C 从机于下列情况下拉低 SCL 时钟（NOSTRETCH = 0）。

- 当址标志被置位：接收到的地址和启用了的从机地址之一匹配上。这种拖延在

ADDR 标志被软件清零后被释放。清零该位的办法是将 ADDRCF 位置 1。

- 在发送时，如果以前的数据传输完毕后，并没有新的数据被写到 I2Cx_TXDR 寄存器，或者如果在 ADDR 标志被清除（TXE=1）后还没有写一个字节，只要数据被写入 I2Cx_TXDR 寄存器，就会释放这个拖延。
- 在接收时如果 I2Cx_RXDR 寄存器的内容还没有被读走，又有一个新的数据被收进来。这时延长在读取 I2Cx_RXDR 时被释放。
- 在从机字节控制模式，重载模式（SBC=1 和 RELOAD=1）时，如果 TCR=1，意味着最后一个数据字节已发送完毕。向 NBYTES[7:0]位域写入一个非零值会清除 TCR 标志，这时延长也会释放。

当 I2Cx_CR1 寄存器的 NOSTRETCH =1 时，I^2C 从机不会延长 SCL 信号（NOSTRETCH=）。

- 在 ADDR 标志置位的时候 SCL 时钟不会延长。
- 发送中，数据必须在对于发送的第一个 SCL 脉冲之前写入 I2Cx_TXDR 寄存器。如果没有，会发生欠载，I2Cx_ISR 寄存器中的 UDR 标志被置位，如果 I2Cx_CR1 寄存器的 ERRIE 位为 1，会产生一个中断。如果首个数据发送已经开始而 STOPF 位还是 1（未被清掉）时，OVR 标志也会被设置。因此，如果你在写下次发送的首个发送数据之后才清除前一次传输的 STOPF 标志，必须先确认 OVR 状态与要发送的第一个数据。
- 在接收时，必须在下一个字节的第 9 个 SCL 脉冲（ACK 脉冲）发生前，从 I2Cx_RXDR 寄存器将数据读走。如果没有被读走，会发生溢出，I2Cx_ISR 寄存器中的 OVR 标志被置位，如果 I2Cx_CR1 寄存器的 ERRIE 位为 1，会产生一个中断。

为了允许从机接收模式的字节 ACK 控制，从机字节控制模式必须通过设置 I2Cx_CR1 寄存器的 SBC 位来启用。这是与 SMBus 标准兼容所必需的。

2．从机发送

当 I2Cx_TXDR 寄存器为空时会产生发送中断状态（TXIS）。如果 I2Cx_CR1 寄存器中的 TXIE 位为 1，则会产生中断。向 I2Cx_TXDR 寄存器写入下一个发送数据时，TXIS 位被清除。

当收到一个 NACK，I2Cx_ISR 寄存器中的 NACKF 位置 1，如果 I2Cx_CR1 寄存器的 NACKIE 位为 1，会产生一个中断。从机会自动释放 SCL 和 SDA 线，这是为了允许主机执行停止或重新启动条件。在收到一个 NACK 时 TXIS 位不会被置 1。

当收到一个 STOP 的时候，I2Cx_ISR 寄存器中的 STOPF 标志会被置 1。如果这时 I2Cx_CR1 寄存器的 STOPIE 又为 1，就会产生一个中断。在大多数应用中，SBC 位通常被设定为 0。在这种情况下，在 TXE 为 0 时收到从机地址（ADDR =1），可以选择是将 I2Cx_TXDR 寄存器的内容当作第一个数据字节发送出去，还是将 TXE 位置 1，从而清空寄存器 I2Cx_TXDR 以便写一个新的数据字节进去。

在从机字节控制模式（SBC= 1），要发送的字节数必须在地址匹配（ADDR=1）中断服务程序中写入 NBYTES。在这种情况下，在传输过程中的 TXIS 事件个数与 NBYTES 中写入的值对应。

注意：当 NOSTRETCH = 1，SCL 时钟在 ADDR 标志被置位的时候不延长，所以不能在 ADDR 子程序中清除 I2Cx_TXDR 寄存器的内容而定义第一个数据字节。要发送的第一个数据字节，必须被预先写到 I2Cx_TXDR 寄存器。

- 这个数据可以是前一次发送消息的时候 TXIS 事件里写入的数据。
- 如果这个数据字节不是要发送的那个，I2Cx_TXDR 寄存器可以随着 TXE 位的置位而被清空，以便写入一个新的数据字节。STOPF 位必须在这些动作之后被清除，以保证它们在第一个数据传输开始前就执行，跟着地址的 ACK。

如果 STOPF 在第一个数据传输开始时仍然为 1，会产生欠载错误（OVR 标志被置 1）。如果需要 1 个 TXIS 事件（发送中断或发送 DMA 请求），除了置 TXE 位外还要置 TXIS 位，这是为了产生 TXIS 的事件。

3．从机接收器

在 I2Cx_RXDR 已有数据时 I2Cx_ISR 寄存器的 RXNE 被置 1，如果 I2Cx_CR1 寄存器的 RXIE 为 1，会产生一个中断。在读取 I2Cx_RXDR 时 RXNE 会被清除。在收到一个 STOP 条件，并且 I2Cx_CR1 的 STOPIE 被置 1，I2Cx_ISR 寄存器的 STOPF 位会被置 1，并产生一个中断。

14.2.6　I²C 主模式

1．I²C 主机初始化

在启动外设之前，必须设置 I2Cx_TIMINGR 寄存器的 SCLH 和 SCLL 的位来配置 I²C 主时钟。这会实施一个时钟同步机制，以支持多主机环境和从机时钟延长。为了让时钟同步：

- 从 SCL 低电平的内部检测开始用 SCLL 计数器来计数时钟低电平的个数。
- 从 SCL 高电平的内部检测开始用 SCLH 计数器来计数时钟高电平的个数。

I²C 依靠 SCL 下降沿的一个 t_{SYNC1} 延迟和 SCL 输入噪声滤波器（模拟+数字）来检测自己的 SCL 低电平，并将 SCL 同步到 I2Cx_CLK 时钟。一旦 SCLL 计数器达到了 I2Cx_TIMINGr 寄存器的 SCLL[7:0]位域中的编程值，I²C 释放 SCL 到高电平。

I²C 依靠 SCL 上升沿的一个 t_{SYNC2} 延迟和 SCL 输入噪声滤波器（模拟+数字）来检测自己的 SCL 高电平，并将 SCL 同步到 I2Cx_CLK 时钟。一旦 SCLH 计数器达到了 I2Cx_TIMINGR 寄存器的 SCLH[7:0]位域中的编程值，I²C 将 SCL 拉低到低电平。

为了启动通信，必须在 I2Cx_CR2 寄存器中设置从机地址的下列参数。

- 地址模式（7 位或 10 位）：ADD10。
- 要发送的从机地址：SADD[9:0]。
- 传输方向：RD_WRN。
- 在 10 位地址的情况下读取 HEAD10R 位。在必须发送完整的地址顺序时，HEAD10R 必须被先置位，以示在方向改变时仅需要帧头的区别。
- 要传输的字节数：NBYTES[7:0]如果字节数大于等于 255 个字节，NBYTES[7:0]最初必须使用 0xff 填充。

随后必须设置 I2Cx_CR2 寄存器的 START 位。START 位一旦被置 1，就不再允许更改上述所有位。只要检测到总线是空闲（BUSY=0）的并插入一个延时后，主机自动发送

START 条件，随后就是发送从机地址。在仲裁丢失的情况下，主机自动切换回从模式，如果从机地址被选中，还将会自动发送 ACK 应答。

2．主机发送器

在写传输的情况下，每个字节发送完之后，TXIS 标志会被置 1，也就是在第 9 个 SCL 脉冲的时候收到一个 ACK。如果 I2Cx_CR1 寄存器的 TXIE 位为 1，则会由 TXIS 事件产生一个中断。向 I2Cx_TXDR 寄存器写入下一个发送数据时，这个标志位被清除。在传输过程中的 TXIS 事件个数与 NBYTES 中写入的值对应。如果数据要发送的字节总数大于 255，必须将 I2Cx_CR2 寄存器的 RELOAD 位置 1 以选择重加载模式。这种情况下，发送了 NBYTES 个字节之后，TCR 标志被置位，并且 SCL 线被拉低直至 NBYTES[7:0]中被写入一个非零值。在收到一个 NACK 时 TXIS 位不会被置 1。

- 当 RELOAD=0 以及 NBYTES 个数据已传输完：
  - 自动结束模式（AUTOEND = 1）下，会自动发送一个 STOP 条件。
  - 在软件结束模式下（AUTOEND = 0），TC 标志被置 1，并且 SCL 线被拉低，这时要软件执行下一步的动作。这时如果已经配置好从机地址和要传送的字节数，就可以将 I2Cx_CR2 中的 START 位置 1，再次发出一个 START 条件。将 START 位置 1 的操作将会清除 TC 标志，并在总线上发出 START 条件。可以将 I2Cx_CR2 寄存器的 STOP 位置 1 来发出停止条件。将 TOP 位置 1 的操作将会清除 TC 标志，并在总线上发出 STOP 条件。
- 如果收到一个 NACK：TXIS 标志不会被置 1，并在收到 NACK 之后自动发送一个 STOP 条件。I2Cx_ISR 寄存器的 NACKF 标志会被置 1，这时如果 NACKIE 位为 1，就会产生中断。

3．主机接收器

在读传输的时，在每个字节接收完后，第 8 个 SCL 脉冲时，RXNE 标志会置 1。如果 I2Cx_CR1 寄存器的 RXIE 位为 1，则会由 RXNE 事件产生一个中断。在读取 I2Cx_RXDR 时 RXNE 会被自动清零。

如果数据要接收的字节总数大于 255，必须将 I2Cx_CR2 寄存器的 RELOAD 位置 1 以选择重加载模式。这种情况下，发送了 NBYTES 个字节之后，TCR 标志被置位，并且 SCL 线被拉低直至 NBYTES[7:0]中被写入一个非零值。

当 RELOAD= 0 以及 NBYTES 个数据已传输完：

- 自动结束模式（AUTOEND = 1）下，在收到最后一个字节后会自动发送一个 NACK 和一个 STOP 条件。
- 软件结束模式（AUTOEND = 0）下，在收到最后一个字节后会自动发送一个 NACK，然后 TC 标志被置 1，并且 SCL 线被拉低，这时要软件来执行后一步的操作。这时如果已经配置好从机地址和要传送的字节数，就可以将 I2Cx_CR2 中的 START 位置 1，再次发出一个 START 条件。将 START 位置 1 的操作将会清除 TC 标志，并在总线上发出 START 条件及从机地址。可以将 I2Cx_CR2 寄存器的 STOP 位置 1 来发出停止条件。将 STOP 位置 1 的操作将会清除 TC 标志，并在总线上发出 STOP 条件。

14.3　I²C 中断

表 14-2 给出了 I²C 中断请求列表。

表 14-2　I²C 中断请求

中断事件	事件标志	清除事件标志/中断	中断使能控制位
读缓冲器非空	RXNE	读 I2Cx_RXDR 寄存器	RXIE
发送缓冲区中断状态	TXIS	写 I2Cx_TXDR 寄存器	TXIE
停止检测中断标志	STOPF	写 STOPCF=1	STOPIE
发送完成重载	TCR	写 I2Cx_CR2 并 NBYTES[7:0]≠0	TCIE
发送完成	TC	写 START=1 或 STOP=1	
地址匹配	ADDR	写 ADDRCF=1	ADDRIE
NACK 接收	NACKF	写 NACKCF=1	NACKIE
总线错误	BERR	写 BERRCF=1	ERRIE
仲裁丢失	ARLO	写 ARLOCF=1	
过载/欠载	OVR	写 OVRCF=1	
PEC 错误	PECERR	写 PECERRCF=1	
超时	TIMEOUT	写 TIMEOUTCF=1	
SMBus 报警	ALERT	写 ALERTCF=1	

图 14-7 中，I²C 的所有这些中断事件共享同一个中断向量（I²C 全局中断）。

使能 I²C 中断的步骤：

（1）在 NVIC 中配置和启用 I²C IRQ 通道。

（2）配置 I²C 以产生中断。

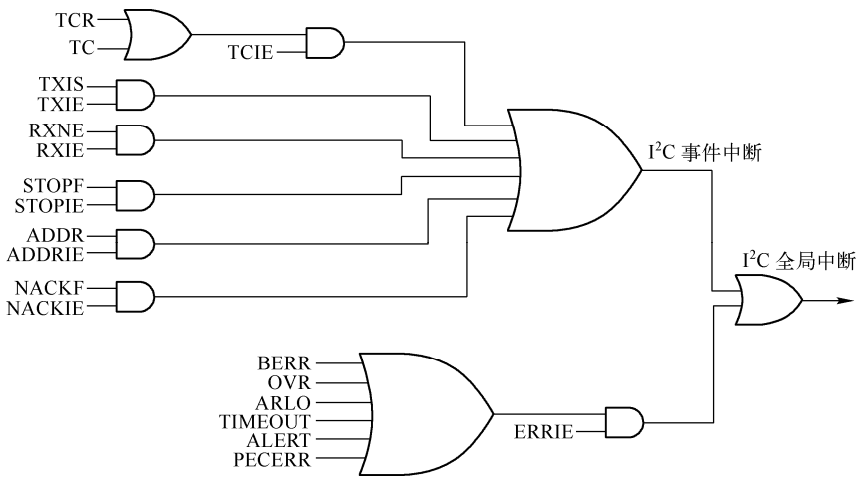


图 14-7　I²C 中断镜像图

14.4 I²C 固件库

表 14-3 为 I²C 的固件库。固件库的使用：

- 使用 RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2Cx, ENABLE)使能 I2C1 或 I2C2 的外设时钟。
- 使用 RCC_AHBPeriphClockCmd()函数使能 SDA、SCL 和 SMBA（被使用时）的 GPIO 时钟。
- 调用 GPIO_Init()函数设置 GPIO 的外设复用功能。
- 通过 I2C_Init()函数设置 I2C 模式、时序、地址、应答和应答地址。
- 下面选项配置无须重新初始化:
  - 通过 I2C_AcknowledgeConfig()函数使能应答特征。
  - 通过 I2C_DualAddressCmd()使能双地址模式。
  - 通过 I2C_GeneralCallCmd()函数使能通用调用。
  - 通过 I2C_StretchClockCmd() 使能时钟延展。
  - 通过 I2C_CalculatePEC() 函数使能 PEC 计算。

通过 I2C_ITConfig()函数使 NVIC 以及相应中断。

- 使用 I2C_Cmd()函数使能 I²C。

表 14-3 I²C 的固件库

函 数	功 能 说 明
void I2C_10BitAddressHeaderCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 10 位头只读方向
void I2C_10BitAddressingModeCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 的主模式下 10 位地址模式
void I2C_AcknowledgeConfig (I2C_TypeDef *I2Cx, FunctionalState NewState)	生成 I²C 通信确认
void I2C_AutoEndCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 自动结束模式（当 nbytes 被传输，停止条件被自动发送）
void I2C_ClearFlag (I2C_TypeDef *I2Cx, uint32_t I2C_FLAG)	清除 I²C 挂起标志
void I2C_ClearITPendingBit (I2C_TypeDef *I2Cx, uint32_t I2C_IT)	清除 I²C 中断挂起标志
void I2C_ClockTimeoutCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 时钟超时（SCL 超时检测）
void I2C_Cmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用指定 I²C 外设
void I2C_DeInit (I2C_TypeDef *I2Cx)	恢复 I2Cx 外设寄存器到默认复位值
void I2C_DMACmd (I2C_TypeDef *I2Cx, uint32_t I2C_DMAReq, FunctionalState NewState)	使能或禁用 I²C 的 DMA 接口
void I2C_DualAddressCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 的地址 2
void I2C_ExtendedClockTimeoutCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 扩展时钟超时（SCL 累积超时检测）
void I2C_GeneralCallCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 通用调用模式
void I2C_GenerateSTART (I2C_TypeDef *I2Cx, FunctionalState NewState)	生成 I2Cx 产生 START 条件
void I2C_GenerateSTOP (I2C_TypeDef *I2Cx, FunctionalState NewState)	生成 I2Cx 通信停止条件
uint8_t I2C_GetAddressMatched (I2C_TypeDef *I2Cx)	返回 I²C 从匹配地址
FlagStatus I2C_GetFlagStatus (I2C_TypeDef *I2Cx, uint32_t I2C_FLAG)	检查特定的 I²C 标识是否被设置

续表

函　数	功 能 说 明
ITStatus　I2C_GetITStatus (I2C_TypeDef *I2Cx, uint32_t I2C_IT)	检查特定的 I²C 中断标识
uint8_t　I2C_GetPEC (I2C_TypeDef *I2Cx)	返回 I²C 的 PEC
uint16_t　I2C_GetTransferDirection (I2C_TypeDef *I2Cx)	返回 I²C 从接收请求
void　I2C_IdleClockTimeoutCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 空闲时钟超时（总线空闲，SCL 和 SDA 高电平测）
void　I2C_Init (I2C_TypeDef *I2Cx, I2C_InitTypeDef *I2C_InitStruct)	根据 I2C_InitStruct 参数配置 I²C 外设
void　I2C_ITConfig (I2C_TypeDef *I2Cx, uint32_t I2C_IT, FunctionalState NewState)	使能或禁用 I²C 中断
void　I2C_MasterRequestConfig (I2C_TypeDef *I2Cx, uint16_t I2C_Direction)	配置主传输请求类型
void　I2C_NumberOfBytesConfig (I2C_TypeDef *I2Cx, uint8_t Number_Bytes)	配置发送/接收的字节数
void　I2C_OwnAddress2Config (I2C_TypeDef *I2Cx, uint16_t Address, uint8_t Mask)	配置 I²C 从地址 2 和屏蔽
void　I2C_PECRequestCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 的 PEC 发送/接收请求
uint32_t　I2C_ReadRegister (I2C_TypeDef *I2Cx, uint8_t I2C_Register)	读取特定的 I²C 寄存器
uint8_t　I2C_ReceiveData (I2C_TypeDef *I2Cx)	返回最近接收到的数据
void　I2C_ReloadCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 的 nbytes 重加载模式
void　I2C_SendData (I2C_TypeDef *I2Cx, uint8_t Data)	通过 I2Cx 发送数据
void　I2C_SlaveAddressConfig (I2C_TypeDef *I2Cx, uint16_t Address)	配置发送的从地址
void　I2C_SlaveByteControlCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 从字节控制
void　I2C_SMBusAlertCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C SMBus 警告
void　I2C_SoftwareResetCmd (I2C_TypeDef *I2Cx)	使能或禁用特定 I²C 软件复位
void　I2C_StopModeCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 从停止模式唤醒
void　I2C_StretchClockCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)	使能或禁用 I²C 时钟延伸
void　I2C_StructInit (I2C_InitTypeDef *I2C_InitStruct)	I2C_InitStruct 配置成默认值
void　I2C_TimeoutAConfig (I2C_TypeDef *I2Cx, uint16_t Timeout)	配置 I²C 总线超时 A
void　I2C_TimeoutBConfig (I2C_TypeDef *I2Cx, uint16_t Timeout)	配置 I²C 总线超时 B
void　I2C_TransferHandling (I2C_TypeDef *I2Cx, uint16_t Address, uint8_t Number_Bytes, uint32_t ReloadEndMode, uint32_t StartStopMode)	当开始传输或者传输过程中处理 I2CX 通信

14.5　读/写 24C02 实例

24C02 是一个有 2Kbit（位）的串行 CMOS EEPROM，即 256 个 8 位字节。另外还有 24C04、24C08、24C16、24C32、24C64 等系列，区别主要在容量、页数、字地址长度等特性。图 14-8 是 24C02 的引脚图。

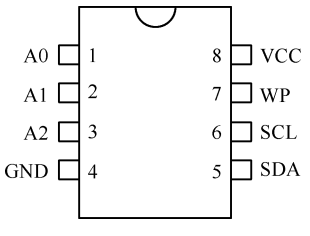


图 14-8　24C02 引脚图

A0，A1，A2：地址输入引脚，走线硬件寻址的依据，同种芯片可同时连接 8 片。

VCC，GND：电源，接地引脚，1.8～5.5V。

WP：写保护。当 WP 接地时，允许对器件的正常读/写操作；当 WP 接高电平时，写保护，只能进行读操作。

SDA：串行地址/数据输入/输出端口，双向传输，漏极开

路，需外接上拉电阻到 VCC（典型阻值为 10kΩ）。

SCL：串行时钟输入，高低电平不同状态与 SDA 配合，执行不同的命令。

I^2C 总线上支持多个从机，图 14-9 是在 STM32F0 的 I^2C 总线挂两个 24C02 的方式。SCL、SDA 信号线分别有 10kΩ的上拉电阻。

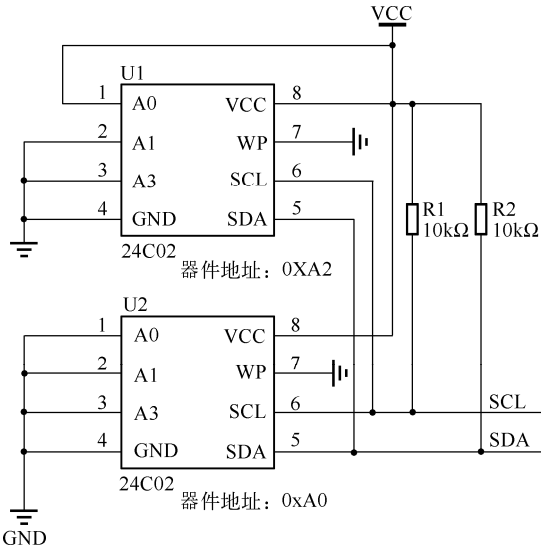


图 14-9　STM32F0 连接两个 24C02

图 14-10 显示了 24C02 的前 8 位是地址信号，从最高位（MSB）开始，其中前 4 位是固定值 1010，后 3 位由管脚 A0、A1、A2 确定。最后一位是读/写控制信号，0 表示写，1 表示读。若与 SDA 线发送过来的地址比较一致，则器件输出应答 0，否则将返回等待状态。所以图 14-9 中的 U2 的地址为 0xA2，U1 的地址为 0xA0。

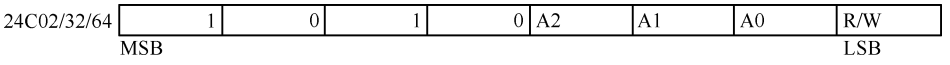


图 14-10　24C02 的器件地址

器件内部地址寻址是在器件寻址之后，对 256 个字节进行寻址，直接传送 8 位地址信号（00～FF）对应于器件内部的地址。

写操作分为两种：一种是字节写，即一次写一个字节；另外一种是页写。

图 14-11 显示了字节写先由主机发送起始命令；再发送送器件地址；主机收到器件的 ACK 应答后发送内部字节地址；收到 ACK 应答后继续发送数据；主机收 ACK 应答，发送停止信号。

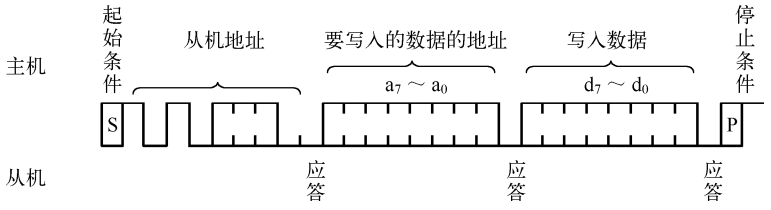


图 14-11　字节写

图 14-12 显示了页的初始化与字节写的初始化相同，只是主器件不会在发送完第一个数据之后就发送停止信号，而是继续发送 7 个数据，接收到每个数据之后，地址的后三位会自动加一，高位地址不变，维持在本页之内；当内部产生的字地址超过了本页的页边界地址时，随后写入的数据将写到该页的页首，先前的字节将会被覆盖。

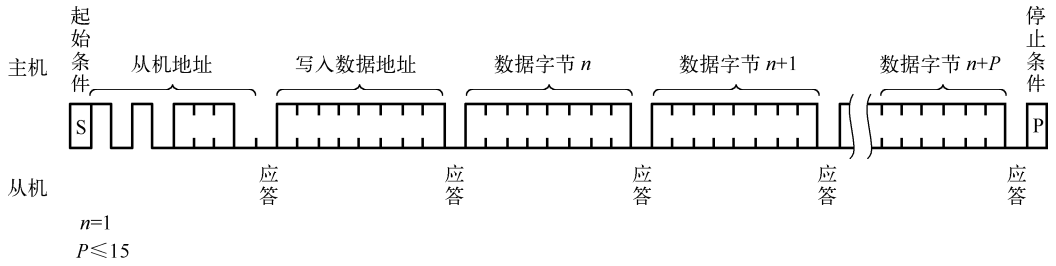


图 14-12　页写

读操作分为当前地址读、随机读和顺序读三种方式。stm320518_eval_i2c_ee.c 中实现的 sEE_ReadBuffer 函数是顺序读。

顺序读（见图 14-13）可以通过随机读或者当前读来启动，主器件接收到一个数据后，应答 ACK；只要从器件接收到 ACK 信号，其字地址自动加 1，并随时钟将数据输出。若到达存储器的末尾，则地址变为 0。如果主器件不发送 ACK 而是停止信号，则结束顺序读。

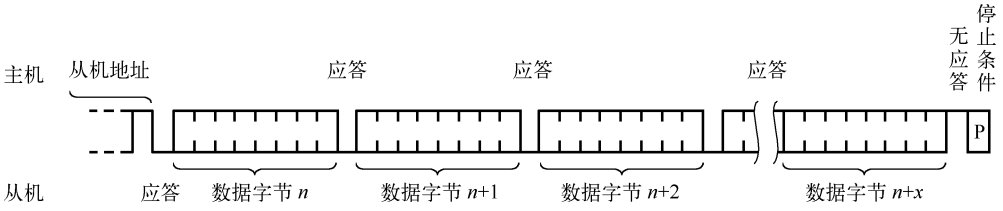


图 14-13　顺序读

在 STM32F0 固件库中已经提供了操作 24C02 的函数，即目录 Utilities\STM32_EVAL\STM320518_EVAL 中的 stm320518_eval_i2c_ee.c 文件。虽然该文件说明是为 ST 公司的 M24LR64 的 EEPROM，但由于 24Cxx 系列的读/写协议均一致，所以该文件支持各厂家的 24Cxx 的 EEPROM。表 14-4 是该文件的函数。sEE_HW_ADDRESS 宏是 EEPROM 的地址。将 EEPROM 分为 sEE_M24C08、sEE_M24C64_32、sEE_M24LR64。其中 sEE_M24C08 支持 24C01、24C02、24C04、24C16。

表 14-4　读/写 24Cxx 函数

函　　数	功　　能
void　sEE_DeInit (void)	释放 I²C EEPROM 外设
void　sEE_Init (void)	初始 I²C EEPROM 外设
uint32_t　sEE_ReadBuffer (uint8_t *pBuffer, uint16_t ReadAddr, uint16_t *NumByteToRead)	从 I²C EEPROM 读取块
uint32_t　sEE_WaitEepromStandbyState (void)	等待 EEPROM 待机状态
void　sEE_WriteBuffer (uint8_t *pBuffer, uint16_t WriteAddr, uint16_t NumByteToWrite)	写缓冲区数据到 I²C EEPROM
uint32_t　sEE_WritePage (uint8_t *pBuffer, uint16_t WriteAddr, uint8_t *NumByteToWrite)	在一个写循环中写入多个字节

注意：sEE_ReadBuffer 函数中 NumByteToRead 参数是以指针形式存在的。

设置好 sEE_HW_ADDRESS，定义 sEE_M24C08 宏，调用读/写 I^2C 的 EEPROM 功能就只有下面几行程序。

```
/* 初始化 I2C EEPROM 驱动器*/
  sEE_Init();
  /*写 I2C EEPROM   */
  sEE_WriteBuffer(Tx1Buffer, sEE_WRITE_ADDRESS1, BUFFER_SIZE1);
  /* 等待 EEPROM 待机状态 */
  sEE_WaitEepromStandbyState();
  /* 设置读的地址 */
  NumDataRead = BUFFER_SIZE1;
  /* 读 I2C EEPROM */
  sEE_ReadBuffer(Rx1Buffer, sEE_READ_ADDRESS1, (uint16_t *)(&NumDataRead));

  /* 验证比较 */
  TransferStatus1 = Buffercmp(Tx1Buffer, Rx1Buffer, BUFFER_SIZE2);
  /* 释放所有占用资源*/
  sEE_DeInit();
```

14.6 小　　结

针对 STM32F1 的 I^2C 功能的瑕疵，STM32F0 的 I^2C 做了重大改动，属于一种全新的设计。STM32F0 在结构、特性、程序接口方面与 STM32F1 都有不同。因此，STM32F0 的 I^2C 程序接口设计与寄存器与 STM32F1 有所不同，STM32F1 上任何有关 I^2C 的代码都要重新编写以便在 STM32F0 上运行。STM32F0 的 I^2C 具有通信事件、由硬件管理、可编程模拟/数字噪声滤波器、独立时钟源（HSI 或 SYSCLK）等新的特性。I^2C 的固件库解决了部分移植问题，在固件库中不仅有文中所谈到 24Cxx 系列的 EEPROM 程序，另外还有 I^2C 接口的 I^2C LM75 温度传感器例程。

第15章

控制器局域网 bxCAN

bxCAN 是基本扩展 CAN（Basic Extended CAN）的缩写，它支持 CAN 协议 2.0A 和 2.0B。CAN 总线是控制器局域网络（Controller Area Network，CAN）的简称，最早由博世公司针对汽车领域开发（相比第 7 章提及的 LIN 网络，CAN 总线昂贵，但性能较好）。在 1993 年标准化（ISO 11898-1）后，CAN 总线从车辆与电子设备领域扩展到航空、航海电子仪器、工厂自动化、工业机械控制、电梯、手扶梯、建筑自动化、医疗仪器与设备等领域。

基于 CAN 总线的上层协议（即应用层）有 CANOpen（工业控制）、DeviceNet（工业控制）、SAE J1939（用于大型汽车）等。CAN 总线上层协议的共同特点是针对 CAN 总线的 11 位标识符（或者 29 位标识符）进行各种形式定义以满足各种需求。

STM32F0 家族只有 STM32F04x、STM32F072 与 STM32F078 具备 CAN 功能。

15.1 bxCAN 概述

CAN 总线属于多主机局域网，采用多主竞争式总线结构，具有多主站运行和分散仲裁以及广播通信的特点，各节点之间可实现自由通信，通信方式灵活。CAN 总线以报文为单位进行数据传送，报文的优先级由标识符决定，具有最小二进制值的标识符具有最高的优先级。CAN 总线采用差分电压传输，在空闲状态下 CAN_H 和 CAN_L 均为 2.5V 左右，此时的状态表示为逻辑“1”，称为“隐性”电平（差值为“0V”）；当 CAN_H 比 CAN_L 高时表示逻辑“0”，称为“显性”电平（差值为“2V”）。显性时，通常电压值为 CAN_H =3.5V，CAN_L=1.5V。

CAN 总线的通信距离可达 10km（5kbps 以下），适用于一些野外场合以及工业现场。

STM32F0x 携带了多种通信外设。针对外设给出一些参考建议：电路板间的通信，可供选择的总线很多，有 USART、SPI、I^2C、CAN 等方式，视芯片资源和成本、软件、通信速率要求而定。在通信距离在几米之内的，常用选择 RS-232、RS-485、USB、CAN、LIN，出于成本以及软件考虑，1.5m 以内通信数据量不大的情况常用 RS-232。但如果通信距离在几米但处于强干扰环境（例如大电流的控制柜）推荐使用 CAN 总线。通信距离在几百米，常用 RS-485、CAN。如果通信距离达到公里级别，CAN 总

线是最佳选择。CAN 总线相比其他方式硬件成本偏高，由于每次发送数据限制 8 个字节以内，如果是数据量偏大但交互较慢的场合，采用 CAN 总线会增加软件工作量。但由于 CAN 总线有抗干扰、节点信息的优先级、报文本身的滤波、CRC 的硬件化等优点，因此其适用范围大大增加。

STM32F0 的 bxCAN 具有支持 CAN 协议 2.0A 和 2.0B 主动模式、波特率最高可达 1Mb/s、支持时间触发通信功能的特点。

发送器具有 3 个发送邮箱，发送报文的优先级特性可软件配置，记录发送 SOF 时刻的时间戳。

接收器具有 3 级深度的 2 个接收 FIFO、14 个过滤组、可配置的 FIFO 溢出处理方，记录接收 SOF 时刻的时间戳。

bxCAN 模块可以完全自动地接收和发送 CAN 报文，且完全支持标准标识符（11 位）和扩展标识符（29 位）。

应用程序通过控制、状态和配置寄存器，可以配置 CAN 参数，如波特率、请求发送报文、处理报文接收、管理中断、获取诊断信息。

共有 3 个发送邮箱供软件发送报文。发送调度器根据优先级决定哪个邮箱的报文先被发送。bxCAN 提供 28 个位宽可变/可配置的标识符过滤器组，软件通过对它们编程，从而在引脚收到的报文中选择它需要的报文，而把其他报文丢弃掉。共有 2 个接收 FIFO，每个 FIFO 都可以存放 3 个完整的报文，它们完全由硬件来管理。bxCAN 有 3 个主要的工作模式：初始化、正常和睡眠模式。在硬件复位后，bxCAN 工作在睡眠模式以节省电能，同时 CANTX 引脚的内部上拉电阻被激活。软件通过对 CAN_MCR 寄存器的 INRQ 或 SLEEP 位置 1，可以请求 bxCAN 进入初始化或睡眠模式。一旦进入了初始化或睡眠模式，bxCAN 就对 CAN_MSR 寄存器的 INAK 或 SLAK 位置 1 来进行确认，同时内部上拉电阻被禁用。当 INAK 和 SLAK 位都为 0 时，bxCAN 就处于正常模式。在进入正常模式前，bxCAN 必须跟 CAN 总线取得同步；为取得同步，bxCAN 要等待 CAN 总线达到空闲状态，即在 CANRX 引脚上监测到 11 个连续的隐性位。

15.2 bxCAN 工作模式

bxCAN 有 8 种工作模式：初始模式、正常模式、低功耗模式、测试模式、静默模式、环回模式、环回静默模式、调试模式。常用的是 3 种工作模式：初始化、正常和睡眠模式。在硬件复位后，bxCAN 工作在睡眠模式以节省电能，CANTX 引脚的内部上拉。通过对 CAN_MCR 寄存器的 INRQ 或 SLEEP 位置 1，bxCAN 进入初始化或睡眠模式。一旦进入了初始化或睡眠模式，bxCAN 就对 CAN_MSR 寄存器的 INAK 或 SLAK 位置 1 来进行确认，同时内部上拉电阻被禁用。当 INAK 和 SLAK 位都为 0 时，bxCAN 就处于正常模式。图 15-1 给出了三种工作模式的转换过程。在进入正常模式前，bxCAN 必须跟 CAN 总线取得同步；为取得同步，bxCAN 要等待 CAN 总线达到空闲状态，即在 CANRX 引脚上监测到 11 个连续的隐性位。

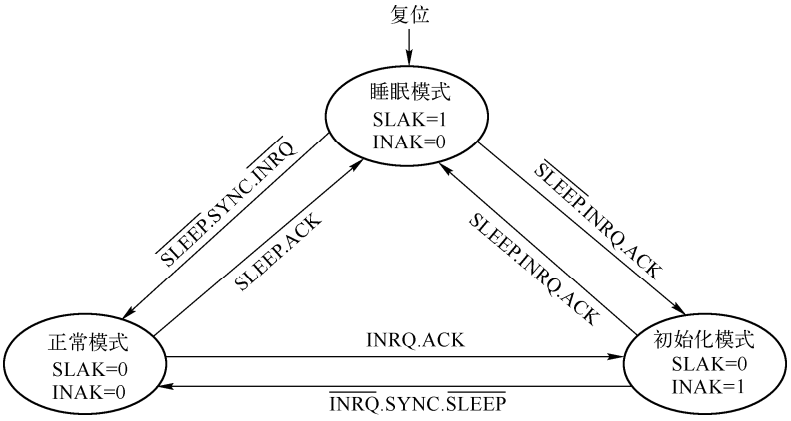


图 15-1　bxCAN 工作模式

15.2.1　初始化模式

软件初始化应该在硬件处于初始化模式时进行。设置 CAN_MCR 寄存器的 INRQ 位为 1，请求 bxCAN 进入初始化模式，然后等待硬件对 CAN_MSR 寄存器的 INAK 位置 1 来进行确认。

清除 CAN_MCR 寄存器的 INRQ 位为 0，请求 bxCAN 退出初始化模式。当硬件对 CAN_MSR 寄存器的 INAK 位清 0 就确认了初始化模式的退出。当 bxCAN 处于初始化模式时，禁止报文的接收和发送，并且 CANTX 引脚输出隐性位（高电平）。初始化模式的进入，不会改变配置寄存器。软件对 bxCAN 的初始化，至少包括位时间特性（CAN_BTR）和控制（CAN_MCR）这 2 个寄存器。

在对 bxCAN 的过滤器组（模式、位宽、FIFO 关联、激活和过滤器值）进行初始化前，软件要对 CAN_FMR 寄存器的 FINIT 位设置 1。对过滤器的初始化可以在非初始化模式下进行。

注意：当 FINIT=1 时，报文的接收被禁止。可以先对过滤器激活位清 0（在 CAN_FA1R 中），然后修改相应过滤器的值。如果过滤器组没有使用，那么就应该让它处于非激活状态（保持其 FACT 位为清 0 状态）。

15.2.2　正常模式

在初始化完成后，软件应该让硬件进入正常模式，以便正常接收和发送报文。软件可以通过对 CAN_MCR 寄存器的 INRQ 位清 0，来请求从初始化模式进入正常模式，然后要等待硬件对 CAN_MSR 寄存器的 INAK 位置 1 的确认。在跟 CAN 总线取得同步，即在 CANRX 引脚上监测到 11 个连续的隐性位（等效于总线空闲）后，bxCAN 才能正常接收和发送报文。

不需要在初始化模式下进行过滤器初值的设置，但必须在它处在非激活状态下完成（相应的 FACTx 位为 0）。而过滤器的位宽和模式的设置，则必须在初始化模式中进入正常模式前完成。

15.2.3 睡眠模式（低功耗）

bxCAN 可工作在低功耗的睡眠模式。软件通过对 CAN_MCR 寄存器的 SLEEP 位置 1，请求进入该模式。睡眠模式下，bxCAN 的时钟停止了，但软件仍然可以访问邮箱寄存器。当 bxCAN 处于睡眠模式，软件必须对 CAN_MCR 寄存器的 INRQ 位置 1 并且同时对 SLEEP 位清 0，才能进入初始化模式。通过软件对 SLEEP 位清 1，或硬件检测到 CAN 总线的活动可以唤醒（退出睡眠模式）bxCAN。如果 CAN_MCR 寄存器的 AWUM 位为 1，一旦检测到 CAN 总线的活动，硬件就自动对 SLEEP 位清 0 来唤醒 bxCAN。如果 CAN_MCR 寄存器的 AWUM 位为 0，软件必须在唤醒中断里对 SLEEP 位清 0 才能退出睡眠状态。

注意：如果唤醒中断被允许（CAN_IER 寄存器的 WKUIE 位为 1），那么一旦检测到 CAN 总线活动就会产生唤醒中断，而不管硬件是否会自动唤醒 bxCAN。

工作模式在对 SLEEP 位清 0 后，睡眠模式的退出必须与 CAN 总线同步。当硬件对 SLAK 位清 0 时，就确认了睡眠模式的退出。

15.2.4 测试模式

初始模式下，通过对 CAN_BTR 寄存器的 SILM 和/或 LBKM 位置 1，来选择一种测试模式。

15.2.5 静默模式

通过对 CAN_BTR 寄存器的 SILM 位置 1，选择静默模式。在静默模式下，bxCAN 可以正常地接收数据帧和远程帧，但只能发出隐性位，而不能真正发送报文。如果 bxCAN 需要发出显性位（确认位、过载标志、主动错误标志），那么这样的显性位在内部被接回来从而被 CAN 内核检测到，同时 CAN 总线不会受到影响而仍然维持在隐性位状态，如图 15-2 所示。因此，静默模式通常用于分析 CAN 总线的活动，而不会对总线造成影响——显性位（确认位、错误帧）不会真正发送到总线上。

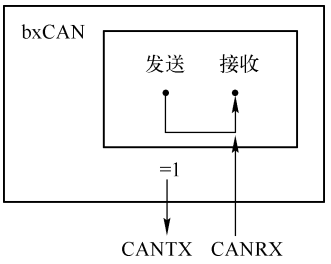


图 15-2 bxCAN 静默模式

15.2.6 环回模式

通过对 CAN_BTR 寄存器的 LBKM 位置 1，来选择环回模式。在环回模式下，bxCAN 把发送的报文当作接收的报文并保存（如果可以通过接收过滤）在接收邮箱里。环回模式可

用于自测试。为了避免外部的影响，在环回模式下 CAN 内核忽略确认错误（在数据/远程帧的确认位时刻，不检测是否有显性位）。在环回模式下，bxCAN 在内部把 Tx 输出反馈到 Rx 输入上，而完全忽略 CANRX 引脚的实际状态，如图 15-3 所示。发送的报文可以在 CANTX 引脚上检测到。

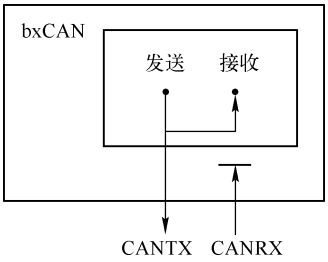


图 15-3　bxCAN 环回模式

15.2.7　环回静默模式

通过对 CAN_BTR 寄存器的 LBKM 和 SILM 位同时置 1，可以选择环回静默模式。该模式可用于“热自测试”，即可以像环回模式那样测试 bxCAN，但却不会影响 CANTX 和 CANRX 所连接的整个 CAN 系统。在环回静默模式下，CANRX 引脚与 CAN 总线断开，同时 CANTX 引脚被驱动到隐性位状态，如图 15-4 所示。

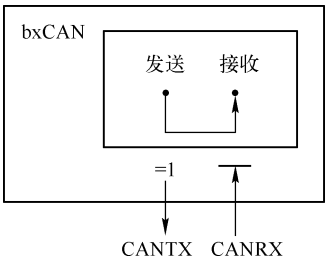


图 15-4　bxCAN 环回静默模式

15.3　bxCAN 功能描述

15.3.1　发送

发送邮箱的状态主要有空置、发送、挂号、预订。图 15-5 给出邮箱发送状态。发送报文的流程：应用程序选择 1 个空置的发送邮箱；设置标识符、数据长度和待发送数据；然后对 CAN_TIxR 寄存器的 TXRQ 位置 1，来请求发送。TXRQ 位置 1 后，邮箱就不再是空邮箱；而一旦邮箱不再为空置，软件对邮箱寄存器就不再有写的权限。TXRQ 位置 1 后，邮箱马上进入挂号状态，并等待成为最高优先级的邮箱。一旦邮箱成为最高优先级的邮箱，其状态就变为预定发送状态。一旦 CAN 总线进入空闲状态，预定发送邮箱中的报文就马上被发

送（进入发送状态）。一旦邮箱中的报文被成功发送后，它马上变为空置邮箱；硬件对 CAN_TSR 寄存器的 RQCP 和 TXOK 位置 1，表明一次成功发送。

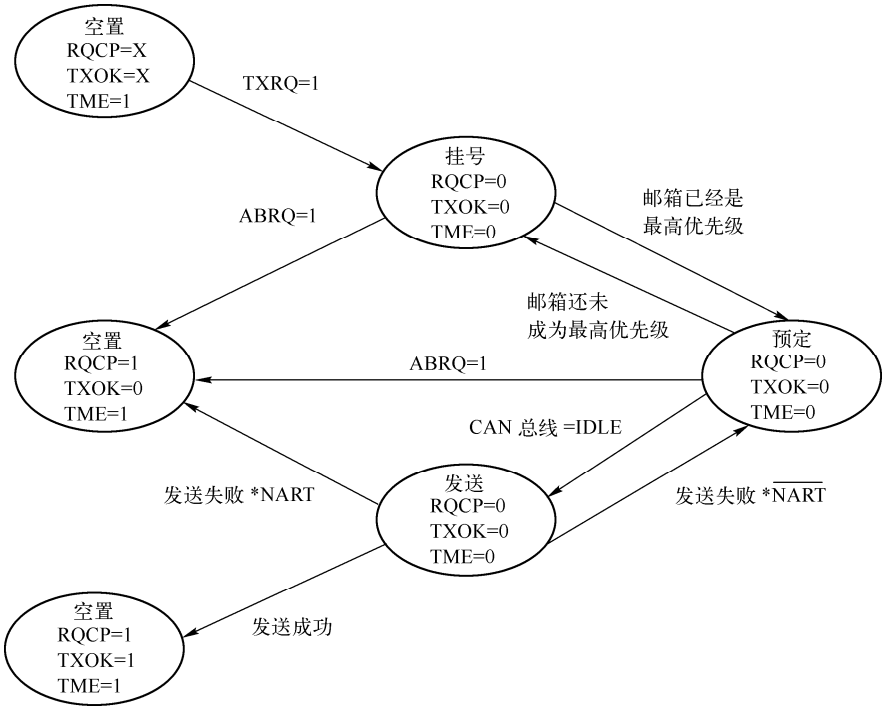


图 15-5　发送邮箱状态

如果发送失败，由于仲裁引起的就对 CAN_TSR 寄存器的 ALST 位置 1，由于发送错误引起的就对 TERR 位置 1。

发送优先级的确定有两种方式：标识符和发送次序。

由标识符决定的，当有超过 1 个发送邮箱在等待时，发送顺序由邮箱中报文的标识符决定。根据 CAN 协议，标识符数值最低的报文具有最高的优先级。如果标识符的值相等，那么邮箱号小的报文先被发送。

由发送请求次序决定的，通过对 CAN_MCR 寄存器的 TXFP 位置 1，可以把发送邮箱配置为发送 FIFO。在该模式下，发送的优先级由发送请求次序决定。该模式对分段发送很有用。

通过对 CAN_TSR 寄存器的 ABRQ 位置 1，可以中止发送请求。邮箱如果处于挂号或预定状态，发送请求马上就被中止了。如果邮箱处于发送状态，那么中止请求可能导致 2 种结果。如果邮箱中的报文被成功发送，那么邮箱变为空置邮箱，并且 CAN_TSR 寄存器的 TXOK 位被硬件置 1。如果邮箱中的报文发送失败了，那么邮箱变为预定状态，然后发送请求被中止，邮箱变为空置邮箱且 TXOK 位被硬件清 0。因此如果邮箱处于发送状态，那么在发送操作结束后，邮箱都会变为空置邮箱。

非自动重发模式主要用于满足 CAN 标准对时间触发通信选项的要求。通过对 CAN_MCR 寄存器的 NART 位置 1，来让硬件工作在该模式。在该模式下，发送操作只会执

行一次。如果发送操作失败了，不管是由于仲裁丢失或出错，硬件都不会再自动发送该报文。在一次发送操作结束后，硬件认为发送请求已经完成，从而对 CAN_TSR 寄存器的 RQCP 位置 1，同时发送的结果反映在 TXOK、ALST 和 TERR 位上。

15.3.2　时间触发通信模式

在时间触发通信模式下，CAN 硬件的内部定时器被激活，并且被用于产生（发送与接收邮箱的）时间戳，分别存储在 CAN_RDTxR/CAN_TDTxR 寄存器中。内部定时器在每个 CAN 位时间累加。内部定时器在接收和发送的帧起始位的采样点位置被采样，并生成时间戳。

15.3.3　接收管理

接收到的报文被存储在 3 级邮箱深度的 FIFO 中。FIFO 由硬件管理，节省了 CPU 的处理负荷，简化了软件并保证了数据的一致性。应用程序只能通过读取 FIFO 输出邮箱，来读取 FIFO 中最先收到的报文。

根据 CAN 协议，当报文被正确接收（直到 EOF 域的最后一位都没有错误），且通过了标识符过滤，那么该报文被认为是有效报文。

FIFO 从空状态开始，在接收到第一个有效的报文后，FIFO 状态变为挂号_1（pending_1），硬件相应地把 CAN_RFR 寄存器的 FMP[1:0]设置为 01（二进制 01b）。软件可以读取 FIFO 输出邮箱来读出邮箱中的报文，然后通过对 CAN_RFR 寄存器的 RFOM 位设置 1 来释放邮箱，这样 FIFO 又变为空状态了。如果在释放邮箱的同时，又收到了一个有效的报文，那么 FIFO 仍然保留在挂号_1 状态，软件可以读取 FIFO 输出邮箱来读出新收到的报文。

如果应用程序不释放邮箱，在接收到下一个有效的报文后，FIFO 状态变为挂号_2（pending_2），硬件相应地把 FMP[1:0]设置为 10（二进制 10b）。重复上面的过程，第三个有效的报文把 FIFO 变为挂号_3 状态（FMP[1:0]＝11b）。此时，软件必须对 RFOM 位设置 1 来释放邮箱，以便 FIFO 可以有空间来存放下一个有效的报文；否则，下一个有效的报文到来时就会导致一个报文的丢失。

当 FIFO 处于挂号_3 状态（即 FIFO 的 3 个邮箱都是满的），下一个有效的报文就会导致溢出，并且一个报文会丢失。此时，硬件对 CAN_RFR 寄存器的 FOVR 位进行置 1 来表明溢出情况。至于哪个报文会被丢弃，取决于对 FIFO 的设置。

如果禁用了 FIFO 锁定功能（CAN_MCR 寄存器的 RFLM 位被清 0），那么 FIFO 中最后收到的报文就被新报文所覆盖。这样，最新收到的报文不会被丢弃掉。

如果启用了 FIFO 锁定功能（CAN_MCR 寄存器的 RFLM 位被置 1），那么新收到的报文就被丢弃，软件可以读到 FIFO 中最早收到的 3 个报文。

一旦往 FIFO 存入一个报文，硬件就会更新 FMP[1:0]位，并且如果 CAN_IER 寄存器的 FMPIE 位为 1，那么就会产生一个中断请求。当 FIFO 变满时（即第 3 个报文被存入），CAN_RFR 寄存器的 FULL 位就被置 1，并且如果 CAN_IER 寄存器的 FFIE 位为 1，那么就会产生一个满中断请求。在溢出的情况下，FOVR 位被置 1，并且如果 CAN_IER 寄存器的 FOVIE 位为 1，那么就会产生一个溢出中断请求。

15.3.4 标识符过滤

在 CAN 协议里，报文的标识符不代表节点的地址，而是跟报文的内容相关的。因此，发送者以广播的形式把报文发送给所有的接收者。节点在接收报文时－根据标识符的值－决定软件是否需要该报文；如果需要，就复制到 SRAM 里；如果不需要，报文就被丢弃且无须软件的干预。为满足这一需求，bxCAN 控制器为应用程序提供了 28 个位宽可变的、可配置的过滤器组（27～0）；在其他产品中，bxCAN 控制器为应用程序提供了 14 个位宽可变的、可配置的过滤器组（13～0），以便只接收那些软件需要的报文。硬件过滤的做法节省了 CPU 开销，否则就必须由软件过滤从而占用一定的 CPU 开销。每个过滤器组 x 由 2 个 32 位寄存器 CAN_FxR0 和 AN_FxR1 组成。图 15-6 给出了过滤器和标识符的对应关系。

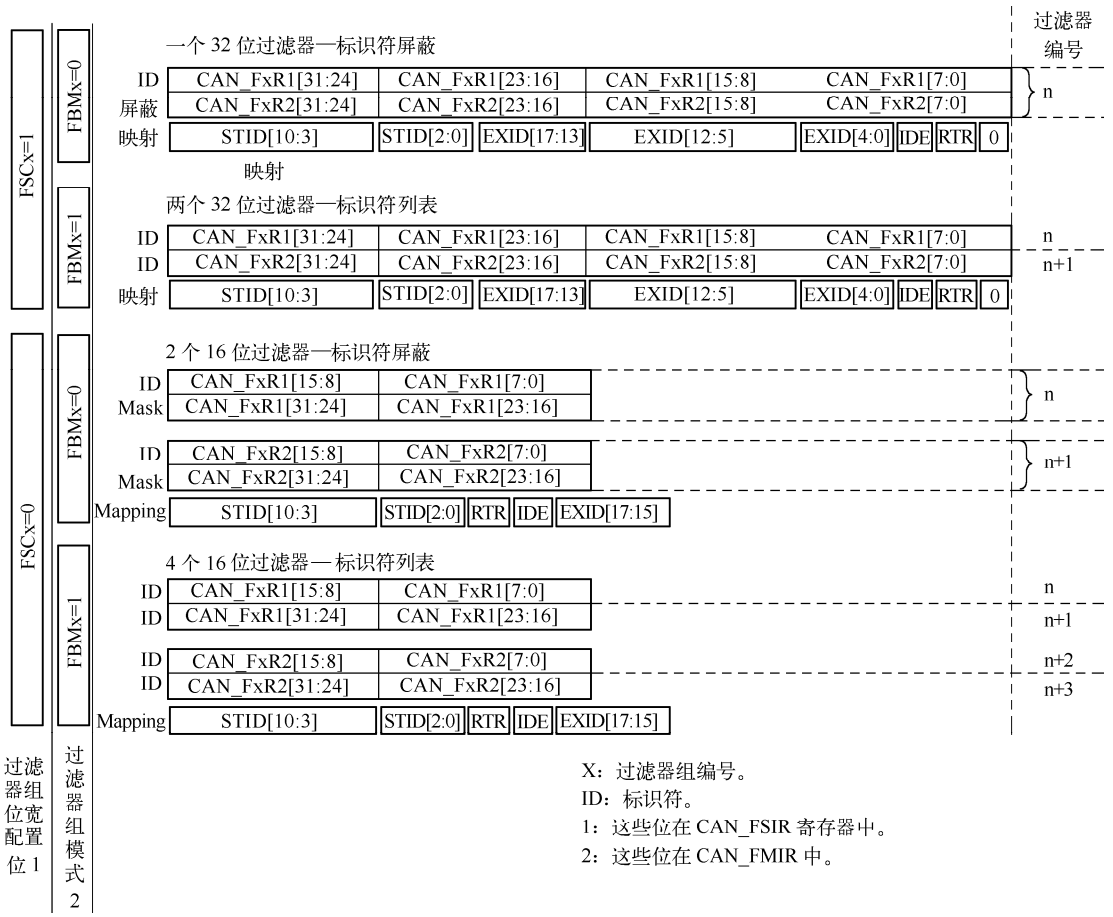


图 15-6　过滤器组位宽设置－寄存器组织

每个过滤器组的位宽都可以独立配置，以满足应用程序的不同需求。根据位宽的不同，每个过滤器组可提供：

- 1 个 32 位过滤器，包括 STDID[10:0]、EXTID[17:0]、IDE 和 RTR 位。
- 2 个 16 位过滤器，包括 STDID[10:0]、IDE、RTR 和 EXTID[17:15]位。

此外过滤器可配置为屏蔽位模式和标识符列表模式。

在屏蔽位模式下，标识符寄存器和屏蔽寄存器一起，指定报文标识符的任何一位，应该按照“必须匹配”或“不用关心”处理。

在标识符列表模式下，屏蔽寄存器也被当作标识符寄存器用。因此，不是采用一个标识符加一个屏蔽位的方式，而是使用 2 个标识符寄存器。接收报文标识符的每一位都必须跟过滤器标识符相同。

过滤器组可以通过相应的 CAN_FMR 寄存器配置。在配置一个过滤器组前，必须通过清除 CAN_FAR 寄存器的 FACT 位，把它设置为禁用状态。通过设置 CAN_FS1R 的相应 FSCx 位，可以配置一个过滤器组的位宽。通过 CAN_FMR 的 FBMx 位，可以配置对应的屏蔽/标识符寄存器的标识符列表模式或屏蔽位模式。为了过滤出一组标识符，应该设置过滤器组工作在屏蔽位模式。为了过滤出一个标识符，应该设置过滤器组工作在标识符列表模式。应用程序不用的过滤器组，应该保持在禁用状态。过滤器组中的每个过滤器，都被编号为（叫做过滤器号）从 0 开始到某个最大数值——取决于过滤器组的模式和位宽的设置。

一旦收到的报文被存入 FIFO，就可被应用程序访问。通常情况下，报文中的数据被复制到 SRAM 中；为了把数据复制到合适的位置，应用程序需要根据报文的标识符来辨别不同的数据。bxCAN 提供了过滤器匹配序号，以简化这一辨别过程。根据过滤器优先级规则，过滤器匹配序号和报文一起，被存入邮箱中。因此每个收到的报文，都有与它相关联的过滤器匹配序号。过滤器匹配序号可以通过下面两种方式来使用：

- 把过滤器匹配序号跟一系列所期望的值进行比较；
- 把过滤器匹配序号当作一个索引来访问目标地址。

对于标识符列表模式下的过滤器（非屏蔽方式的过滤器），软件不需要直接跟标识符进行比较。对于屏蔽位模式下的过滤器，软件只须对需要的那些屏蔽位（必须匹配的位）进行比较即可。在给过滤器编号时，并不考虑过滤器组是否为激活状态。另外，每个 FIFO 各自对其关联的过滤器进行编号。

根据过滤器的不同配置，有可能一个报文标识符能通过多个过滤器的过滤；在这种情况下，存放在接收邮箱中的过滤器匹配序号，根据下列优先级规则来确定：

- 位宽为 32 位的过滤器，优先级高于位宽为 16 位的过滤器；
- 对于位宽相同的过滤器，标识符列表模式的优先级高于屏蔽位模式；
- 位宽和模式都相同的过滤器，优先级由过滤器号决定，过滤器号小的优先级高。

15.3.5　报文存储

邮箱是软件和硬件之间传递报文的接口。邮箱包含了所有跟报文有关的信息：标识符、数据、控制、状态和时间戳信息。

软件需要在一个空的发送邮箱中，把待发送报文的各种信息设置好（然后再发出发送的请求）。发送的状态可通过查询 CAN_TSR 寄存器获知。

在接收到一个报文后，软件就可以访问接收 FIFO 的输出邮箱来读取它。一旦软件处理了报文（如把它读出来），软件就应该对 CAN_RFxR 寄存器的 RFOM 位进行置 1，来释放该报文，以便为后面收到的报文留出存储空间。过滤器匹配序号存放在 CAN_RDTxR 寄存器的 FMI 域中。16 位的时间戳存放在 CAN_RDTxR 寄存器的 TIME[15:0]域中。

15.3.6 错误管理

由硬件通过发送错误计数器（CAN_ESR 寄存器里的 TEC 域）和接收错误计数器（CAN_ESR 寄存器里的 REC 域）来实现 CAN 协议中的出错管理。软件可以读出它们的值来判断 CAN 网络的稳定性。此外，CAN_ESR 寄存器提供了当前错误状态的详细信息。通过设置 CAN_IER 寄存器（比如 ERRIE 位），当检测到出错时软件可以灵活地控制中断的产生。当 TEC 大于 255 时，bxCAN 就进入离线状态，同时 CAN_ESR 寄存器的 BOFF 位被置 1。在离线状态下，bxCAN 无法接收和发送报文。根据 CAN_MCR 寄存器中 ABOM 位的设置，bxCAN 可以自动或在软件的请求下，从离线状态恢复（变为错误主动状态）。在这两种情况下，bxCAN 都必须等待一个 CAN 标准所描述的恢复过程（CAN RX 引脚上检测到 128 次 11 个连续的隐性位）。如果 ABOM 位为 1，bxCAN 进入离线状态后，就自动开启恢复过程。如果 ABOM 位为 0，软件必须先请求 bxCAN 进入然后再退出初始化模式，随后恢复过程才被开启。图 15-7 给出 CAN 错误状态。

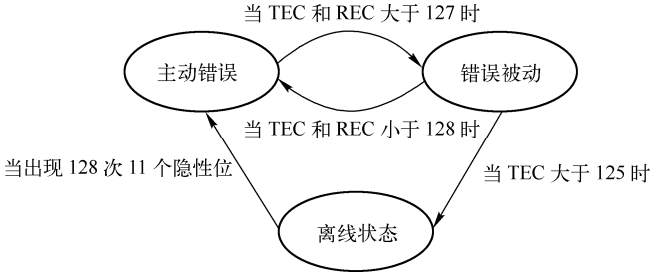


图 15-7　CAN 错误状态图

注意：*在初始化模式下，bxCAN 不会监视 CAN RX 引脚的状态，这样就不能完成恢复过程。为了完成恢复过程，bxCAN 必须工作在正常模式。*

15.3.7 位时间特性

位时间特性逻辑通过采样来监视串行的 CAN 总线，并且通过与帧起始位的边沿进行同步，以及通过与后面的边沿进行重新同步，来调整其采样点。每位时间分为 3 段，如下所述。

- 同步段（SYNC_SEG）：通常期望位的变化发生在该时间段内。其值固定为 1 个时间单元（$1\times t_{CAN}$）。
- 时间段 1（BS1）：定义采样点的位置。它包含 CAN 标准里的 PROP_SEG 和 PHASE_SEG1。其值可以编程为 1～16 个时间单元，但也可以被自动延长，以补偿因为网络中不同节点的频率差异所造成的相位的正向漂移。
- 时间段 2（BS2）：定义发送点的位置。它代表 CAN 标准里的 PHASE_SEG2。其值可以编程为 1～8 个时间单元，但也可以被自动缩短以补偿相位的负向漂移。

重新同步跳跃宽度（SJW）定义了，在每位中可以延长或缩短多少个时间单元的上限。其值可以编程为 1～4 个时间单元。有效跳变被定义为，当 bxCAN 自己没有发送隐性位时，

从显性位到隐性位的第 1 次转变。如果在时间段 1（BS1）而不是在同步段（SYNC_SEG）检测到有效跳变，那么 BS1 的时间就被延长最多 SJW 那么长，从而采样点被延迟了。相反如果在时间段 2（BS2）而不是在 SYNC_SEG 检测到有效跳变，那么 BS2 的时间就被缩短最多 SJW 那么长，从而采样点被提前了。

为了避免软件的编程错误，对位时间特性寄存器（CAN_BTR）的设置，只能在 bxCAN 处于初始化状态下进行。

15.4　bxCAN 中断

bxCAN 占用 4 个专用的中断向量。通过设置 CAN 中断允许寄存器（CAN_IER），每个中断源都可以单独允许和禁用。

- 发送中断可由下列事件产生：
  - 发送邮箱 0 变为空，CAN_TSR 寄存器的 RQCP0 位被置 1。
  - 发送邮箱 1 变为空，CAN_TSR 寄存器的 RQCP1 位被置 1。
  - 发送邮箱 2 变为空，CAN_TSR 寄存器的 RQCP2 位被置 1。
- FIFO0 中断可由下列事件产生：
  - FIFO0 接收到一个新报文，CAN_RF0R 寄存器的 FMP0 位不再是 00。
  - FIFO0 变为满的情况，CAN_RF0R 寄存器的 FULL0 位被置 1。
  - FIFO0 发生溢出的情况，CAN_RF0R 寄存器的 FOVR0 位被置 1。
- FIFO1 中断可由下列事件产生：
  - FIFO1 接收到一个新报文，CAN_RF1R 寄存器的 FMP1 位不再是 00。
  - FIFO1 变为满的情况，CAN_RF1R 寄存器的 FULL1 位被置 1。
  - FIFO1 发生溢出的情况，CAN_RF1R 寄存器的 FOVR1 位被置 1。
- 错误和状态变化中断可由下列事件产生：
  - 出错情况，关于出错情况的详细信息请参考 CAN 错误状态寄存器（CAN_ESR）。
  - 唤醒情况，在 CAN 接收引脚上监视到帧起始位（SOF）。
  - CAN 进入睡眠模式。

15.5　bxCAN 固件库

表 15-1 是 CAN 的固件库函数列表。固件库的使用如下。

- 通过 RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN, ENABLE)使能 CAN 控制器接口时钟。
- 通过 CC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOx, ENABLE)设置 CAN 对应的 GPIO 引脚时钟；GPIO_PinAFConfig(GPIOx, GPIO_PinSourcex, GPIO_AF_CANx)设置 CAN 引脚的复用功能；调用 GPIO_Init()函数配置 CAN 引脚。
- 使用 CAN_Init()和 CAN_FilterInit()函数初始化 CAN 和 CAN 过滤器。
- 通过 CAN_Transmit()函数发送 CAN 帧。

表 15-1　CAN 固件库

组	函　数	描　述
设置默认状态	void CAN_DeInit(CAN_TypeDef* CANx)	恢复 CAN 寄存器到默认值
初始化和配置函数	uint8_t CAN_Init(CAN_TypeDef* CANx, CAN_InitTypeDef* CAN_InitStruct)	由输入参数初始化 CAN 外设
	void CAN_FilterInit(CAN_FilterInitTypeDef* CAN_FilterInitStruct)	初始化 CAN 过滤器
	void CAN_StructInit(CAN_InitTypeDef* CAN_InitStruct)	设置 CAN_InitStruct 为默认值
	void CAN_SlaveStartBank(uint8_t CAN_BankNumber)	设置从 CAN 的过滤组
	void CAN_DBGFreeze(CAN_TypeDef* CANx, FunctionalState NewState)	使能或禁用 CAN 调试暂停功能
	void CAN_TTComModeCmd(CAN_TypeDef* CANx, FunctionalState NewState)	使能或禁用 CAN 时间触发通信模式
CAN 发送帧	uint8_t CAN_Transmit(CAN_TypeDef* CANx, CanTxMsg* TxMessage)	发送 CAN 帧
	uint8_t CAN_TransmitStatus(CAN_TypeDef* CANx, uint8_t TransmitMailbox)	检查帧发送状态
	void CAN_CancelTransmit(CAN_TypeDef* CANx, uint8_t Mailbox)	取消发送
CAN 接收帧	void CAN_Receive(CAN_TypeDef* CANx, uint8_t FIFONumber, CanRxMsg* RxMessage)	接收帧
	void CAN_FIFORelease(CAN_TypeDef* CANx, uint8_t FIFONumber)	释放接收 FIFO
	uint8_t CAN_MessagePending(CAN_TypeDef* CANx, uint8_t FIFONumber)	返回挂起的接收消息数
运行模式	uint8_t CAN_OperatingModeRequest(CAN_TypeDef* CANx, uint8_t CAN_OperatingMode)	选择 CAN 的操作模式
	uint8_t CAN_Sleep(CAN_TypeDef* CANx)	进入低功耗模式
	uint8_t CAN_WakeUp(CAN_TypeDef* CANx)	从低功耗模式唤醒
CAN 总线错误管理	uint8_t CAN_GetLastErrorCode(CAN_TypeDef* CANx)	返回最后故障码
	uint8_t CAN_GetReceiveErrorCounter(CAN_TypeDef* CANx)	返回 CAN 接收错误计数
	uint8_t CAN_GetLSBTransmitErrorCounter(CAN_TypeDef* CANx)	返回 9 位 CAN 发送错误计数的最低有效位
中断和标志管理	void CAN_ITConfig(CAN_TypeDef* CANx, uint32_t CAN_IT, FunctionalState NewState)	使能或禁用特定的 CAN 中断
	FlagStatus CAN_GetFlagStatus(CAN_TypeDef* CANx, uint32_t CAN_FLAG)	读取标志
	void CAN_ClearFlag(CAN_TypeDef* CANx, uint32_t CAN_FLAG)	清除特定标志
	ITStatus CAN_GetITStatus(CAN_TypeDef* CANx, uint32_t CAN_IT)	读取特定的中断状态
	void CAN_ClearITPendingBit(CAN_TypeDef* CANx, uint32_t CAN_IT)	返回特定的中断状态

- 调用 CAN_TransmitStatus()函数检查发送帧状态。
- 可通过 CAN_CancelTransmit()函数取消 CAN 帧的发送。
- 通过 CAN_Recieve()读取接收到数据。
- 通过 CAN_FIFORelease()释放接收 FIFO。
- CAN_MessagePending()查询挂起的接收帧数量。
- 通过 CAN_GetFlagStatus()函数轮询方式或者 CAN_ITConfig()与 CAN_GetITStatus()中断方式控制 CAN 事件。

15.6　CAN 通信实例

由于 STM32F03X 系列无 CAN 外设，本节实例基于 STM32F072 Discovery（STM32F072RBT6）实现。在 DM00090510 文档中给出了 STM32F072RBT6 对应的 CAN 引脚：

PA11、PA12（AF4）与 PB8、PB9（AF4）分别对应 CAN_Rx 与 CAN_TX 功能。

程序中选用了 PA11 与 PA12。对应设置在 main.h 文件中，在此设置了引脚以及引脚对应时钟、引脚复用。

```
#define CANx                        CAN
#define CAN_CLK                     RCC_APB1Periph_CAN
#define CAN_RX_PIN                  GPIO_Pin_11
#define CAN_TX_PIN                  GPIO_Pin_12
#define CAN_GPIO_PORT               GPIOA
#define CAN_GPIO_CLK                RCC_AHBPeriph_GPIOA
#define CAN_AF_PORT                 GPIO_AF_4
#define CAN_RX_SOURCE               GPIO_PinSource11
#define CAN_TX_SOURCE               GPIO_PinSource12
```

采用了环回方式，即自发自收方式。程序中采用了 29 位 ID，接收部分仅接收与高 8 位匹配的 ID，参见图 15-6。

```
int main(void)
{
  /* CAN 配置 */
  CAN_Config();

  while(1)
  {
        TxMessage.Data[0] = 0x1F;
        CAN_Transmit(CANx, &TxMessage);
        Delay();
  }
}

/* 配置 CAN */
static void CAN_Config(void)
{
  GPIO_InitTypeDef    GPIO_InitStructure;
  NVIC_InitTypeDef    NVIC_InitStructure;
  CAN_InitTypeDef           CAN_InitStructure;
  CAN_FilterInitTypeDef    CAN_FilterInitStructure;

  /* CAN GPIOs 配置 */

  /* 使能 GPIO 时钟 */
```

```
RCC_AHBPeriphClockCmd(CAN_GPIO_CLK, ENABLE);

/* CAN 引脚为 AF4 */
GPIO_PinAFConfig(CAN_GPIO_PORT, CAN_RX_SOURCE, CAN_AF_PORT);
GPIO_PinAFConfig(CAN_GPIO_PORT, CAN_TX_SOURCE, CAN_AF_PORT);

/*  配置 CAN Rx 和 Tx 引脚  */
GPIO_InitStructure.GPIO_Pin = CAN_RX_PIN | CAN_TX_PIN;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;
GPIO_Init(CAN_GPIO_PORT, &GPIO_InitStructure);

/* NVIC  配置  */
NVIC_InitStructure.NVIC_IRQChannel = CEC_CAN_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPriority = 0x0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);

/* CAN 配置  */
/*  使能 CAN 时钟  */
RCC_APB1PeriphClockCmd(CAN_CLK, ENABLE);

/* CAN 寄存器初始化 t */
CAN_DeInit(CANx);
CAN_StructInit(&CAN_InitStructure);

/* CAN  单元初始化  */
CAN_InitStructure.CAN_TTCM = DISABLE;
CAN_InitStructure.CAN_ABOM = DISABLE;
CAN_InitStructure.CAN_AWUM = DISABLE;
CAN_InitStructure.CAN_NART = DISABLE;
CAN_InitStructure.CAN_RFLM = DISABLE;
CAN_InitStructure.CAN_TXFP = DISABLE;
CAN_InitStructure.CAN_Mode = CAN_Mode_LoopBack; //环回模式，正常模式为 CAN_Mode_Normal;
CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;

/* CAN  波特率  = 1MBps (CAN 时钟为 36MHz) */
CAN_InitStructure.CAN_BS1 = CAN_BS1_9tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_8tq;
CAN_InitStructure.CAN_Prescaler = 2;
CAN_Init(CANx, &CAN_InitStructure);

/* CAN 滤波器设置  */
CAN_FilterInitStructure.CAN_FilterNumber = 0;//选择过滤器 0
CAN_FilterInitStructure.CAN_FilterMode = CAN_FilterMode_IdMask;//标识符屏蔽位模式
CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_32bit;//32 位过滤器
```

```
    CAN_FilterInitStructure.CAN_FilterIdHigh  =0x8000;//扩展为 0x8000;如发送的是标准 ID 则为(0x321<<5);
    CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh  =0xFF00; //只接收 StdID（或扩展 ID）的高 8 位匹配的 ID
    CAN_FilterInitStructure.CAN_FilterMaskIdLow =0x0000;
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment = 0;//选择 FIFO0
    CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;//使能 CAN 过滤器
    CAN_FilterInit(&CAN_FilterInitStructure);

    /* 发送结构体配置 */
    TxMessage.StdId = 0x321;//
    TxMessage.ExtId =0x10000000;          //扩展 Id，滤波器验证第 29 位
    TxMessage.RTR =CAN_RTR_DATA;
    TxMessage.IDE = CAN_ID_EXT;           //扩展 Id 为 CAN_ID_EXT，即 29 位 Id，标准 Id 则为 CAN_ID_ STD；二者取一
    TxMessage.DLC = 1;

    /* 使能 FIFO 0 消息挂起中断*/
    CAN_ITConfig(CANx, CAN_IT_FMP0, ENABLE);
  }
```

中断函数：

```
void CEC_CAN_IRQHandler(void)
{
  CAN_Receive(CANx, CAN_FIFO0, &RxMessage);
}
```

图 15-8 是作者经常用到的 CAN 收发器外设电路。图中 CAN_RD 和 CAN_TD 连接 STM32F072RBT6 中的相应引脚，如 PA11 和 PA12。

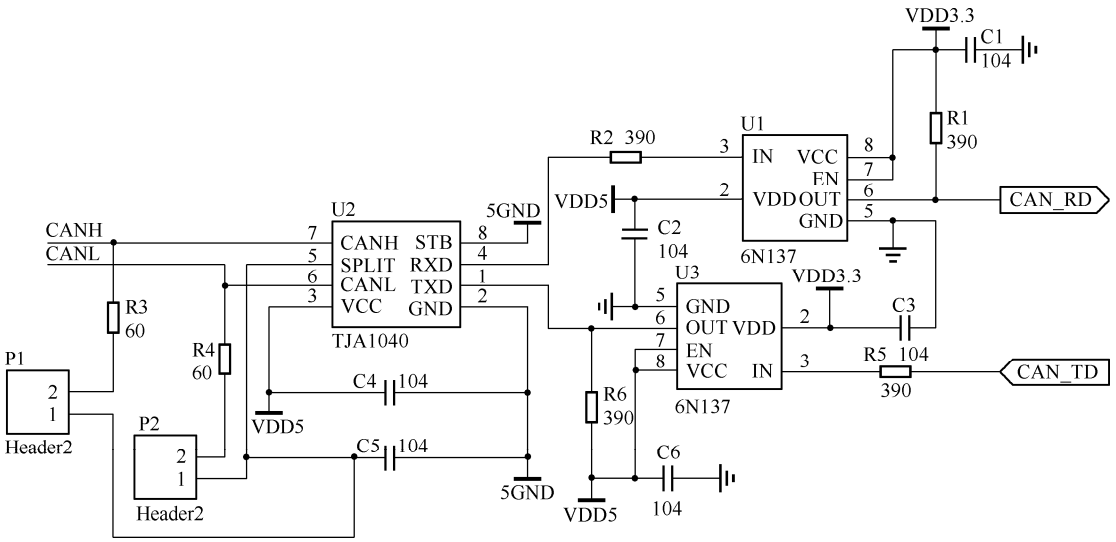


图 15-8　CAN 外设收发器

图中 R3 和 R4 构成匹配电阻 120Ω，对于距离比较近的两个 CAN 节点，断开 P1 和 P2 的跳线。对于距离比较远和多个节点情况，参照图 15-9 和图 15-10，只是在首尾两个 CAN 节点使用跳线帽短接 P1 和 P2，其余节点断开 P1 和 P2，否则 CAN 通信不正常。

TJA1040 收发器提供了 CAN 控制器与物理总线之间的接口以及对 CAN 总线的差动发送和接收功能。芯片内部有一个电流限制电路和一个温度保护电路。可通过引脚 STB 选择工作模式：高速或待机模式。STB 引脚接地即为高速模式，不连接则默认为高速模式。芯片内部有一个超时定时器，用来检测 TXD 引脚的低电平。

其中 6N137 属于高速光耦，转换速率高达 10Mbit/s，起到电气隔离作用，保护控制系统电路。

图中 VDD5 与 5GND 代表 5V 与 5V 的地；VDD3.3 代表 3.3V，对应的地与 5V 地采用一点共地。

对于如何用或者说如何将 29 位 ID（或 11 位 ID）与远程帧（数据帧）更好地发挥作用，不妨参考一下 CANOpen 协议或者 DeviceNet 协议的设计。CANOpen 应用层协议将标准报文的 11 位标识符中高 4 位定义为功能码，剩余 7 位为节点号，即通信对象标识符。7 位节点号则表明 CANOpen 网络最多支持 127 个节点（0 号为主节点）。DeviceNet 的标识符分配方案是面向节点的信息标识符分配，对于 DeviceNet 系统，最多数量为 64 个的节点，其每一个节点拥有一组出自于 3 个信息组的标识符。

作者常用的格式是 29 位标志符：28～21 位为目标节点地址；20～13 位发送节点地址；12～9 位代表设备类型；8～5 位表示命令类型，即读/写或者读/写响应；4～0 位代表数据位是否多于 8 位数据，应用层根据此 4 位决定将收到数据的拼装方法。该方式缺陷是定义的节点数不能超过 256 个。

将 CAN 总线应用工业现场，必然涉及通信介质，但 CAN 总线本身对通信介质无明确规定，可以是双绞线、同轴电缆、光纤。由于屏蔽双绞线的成本低以及便捷特点，工业现场中多采用单屏蔽层的双绞线。一般原则要求屏蔽双绞线单点接地。但对于传输距离较远情况，可采用屏蔽层分段接地。

CAN 总线的网络拓扑结构较为常见的是总线型，即图 15-9 的“直线式”的形式；相比星形或环形网络，网络有两个“端点”。在两个端点上，都有 1 个大约 120Ω 的终端电阻被连接在 CAN_H 和 CAN_L 信号线上。

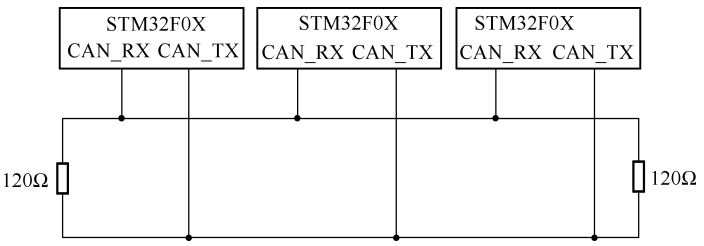


图 15-9　总线型 CAN 拓扑

但由于分支长度以及分支长度的积累都会造成阻抗不连续，在接头处产生“反射”现象，所以直线型拓扑结构中，最常用的是图 15-10 中手拉手式的连接。

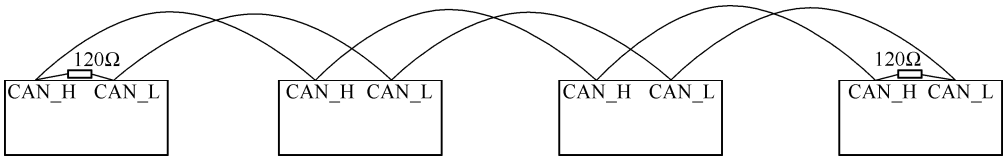


图 15-10　总线型 CAN 的手拉手连线图

15.7　小　　结

CAN 总线由于实时性强、传输距离远、抗干扰能力强、成本低等优点，已经从最开始的汽车领域扩展到各种工业控制领域。STM32F0x 家族的 STM32F04x、STM32F072 与 STM32F078 拥有 CAN 外设，支持 CAN2.0A 和 CAN2.0B，但 CAN 总线的使用需要结合工业现场环境，选用相应的通信介质、标识符使用规则、相应的总线拓扑结构、故障处理机制。在 15.6 节给出了一些设计参考。

第16章

RTX 实时操作系统应用

第 4 章提及嵌入式应用程序程序结构较为常用的是中断与轮询混合方式，即以中断为核心多个任务顺序执行的方式。但当任务比较多，任务之间切换比较频繁，并且任务之间要求平行性，使用裸机方式就显得力不从心。实时操作系统可保证系统的实时性、将复杂的系统分解为相对独立的多个任务，达到“分而制之”的目的，从而降低系统的复杂性。本章将讲解关于 ARM 公司的 RTX，即 MDK 中包含的 RTX 内核。RTX 内核是一个实时操作系统（RTOS），RTX 负责创建任务；开始执行任务和停止任务；可运行多个任务；自由调度系统资源；并通过信号量、时间、邮箱、互斥实现任务之间的通信以及资源的管理。

16.1 RTX 概述

在中断与超级循环混合方式的嵌入式系统中，中断服务程序（ISR）起关键作用。但如需要在中断服务例程中处理各种运算，将使 ISR 变得复杂，而且执行时间较长。while(1)内的函数与 ISR 之间的数据交互是通过全局共享变量进行的，但 RTOS 是通过任务调度实现任务管理，从而确保了更好的程序流和事件响应。通过事件、信号量、邮箱的任务通信可将一些原中断内的功能转移到中断服务程序外的高优先级任务中，从而缩短中断服务程序的处理时间。RTX 是包含在 MDK 软件中的一款实时操作系统，具有如下特点。

- 灵活的调度：循环、抢先和协作。
- 以低的中断延迟执行高速实时操作。
- 占用空间小，适用于资源受限的系统。
- 不限数量的任务，每个任务都具有 254 个优先级。
- 不限数量的邮箱、信号、互斥函数和计时器。
- 支持多线程和线程安全运算。
- MDK-ARM 中的内核识别调试支持（需芯片支持，STM32F0 不支持）。
- 使用 μVision 配置向导的基于对话框的设置。

实时操作系统一个较大特点是确定性，即运行时间的确定性，但并非每个 RTOS 都具有确定性。RTX 提供完全确定性的行为，这意味着在预定义时间内（期限）处理事件和中断。应用程序可以依赖于一致且已知的任务计时。

RTX 的定义任务数量无任何限制，活动任务最大为 250，可在 RTX_CONFIG.C 文件中

定义。任务优先级是 1～254。支持堆栈检查功能。RTX 的任务之间通信是通过事件、信号量、互斥、邮箱如图 16-1 所示。其中事件伴随任务创建而创建，每个任务有 16 个事件标志。邮箱的大小不受限制。表 16-1 是 Keil 提供的 RTX 技术参数。

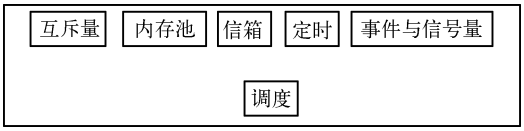


图 16-1　RTX 结构图

用于 Cortex-M 的 RTX 内核库不禁用中断。ISR 的中断响应时间与无 RTX 内核时相同。这个是与其他操作系统的处理机制不同之处。换句话说，前面章节中提到的所有中断例程可原封不动使用。

目前 RTX 的源代码已经包含在 MDK 软件中，方便学习、研究。MDK5 相比原有的 MDK4 有很多变化，ARM 公司为了保证向下兼容，在 http://www2.keil.com/mdk5/legacy/提供了兼容方案。

表 16-1　RTX 的技术参数

描述	ARM7™/ARM9™	Cortex™-M
定义任务	无限制	无限制
活动任务	最大 250	最大 250
信箱	无限制	无限制
信号量	无限制	无限制
互斥量	无限制	无限制
信号/事件	每任务 16 个	每任务 16 个
用户定时器	无限制	无限制
代码空间	<4.2KB	<4.0KB
内核的 RAM 空间	300B +80B 用户栈	300B +128B 栈
任务的 RAM 空间	任务堆栈+52B	任务堆栈+52B
信箱的 RAM 空间	MaxMessages * 4 + 16B	MaxMessages * 4 + 16B
信号量的 RAM 空间	8B	8B
互斥量的 RAM 空间	12B	12B
用户定时器的 RAM 空间	8B	8B
硬件需求	片上定时器	SysTick 时钟
用户任务优先级	1～254	1～254
任务切换时间	<5.3μs@ 60MHz	<2.6μs@72MHz
中断停止时间	<2.7μs@ 60MHz	RTX 不禁用中断

16.1.1　RTX 任务

任务是一个简单的程序，应用程序的每个作业由分割的任务处理，每个外设也可由几个分割的任务处理。每个任务被赋予一定的优先级，范围是 1～254，其中 254 是最高优先级，

1 是最低优先级，0 为空闲任务保留。如果创建用户任务时，赋值 0 优先级，自动转为优先级 1。需要注意的是任务优先级与中断优先级无关联。在 RTX 内核，每个任务的状态可能是表 16-2 中的一种状态。

表 16-2　任务状态

状态		描　述
运行（RUNNING）		表明当前的任务处于运行态。在任意时刻，只能有一个任务处于该状态。函数 os_tsk_self()返回当前运行任务的任务号（TID）
就绪（READY）		表明任务处于就绪状态。一旦运行的任务处理完毕，RTX 核选择下一个具有最高优先级的任务开始运行
等待状态	WAIT_DLY	等待延迟超时的任务处于 WAIT_DLY 状态。一旦延迟超时，该任务切换到就绪状态。函数 os_dly_wait()可以使任务处于 WAIT_DLY 状态
	WAIT_ITV	等待间隔超时的任务处于 WAIT_ITV 状态。一旦间隔超时，任务切换到就绪状态。函数 os_itv_wait()可以使任务处于 WAIT_IVL 状态
	WAIT_OR	等待最少一个事件标志的任务处于 WAIT_OR 状态。一旦获得事件标志，任务转换到就绪状态。函数 os_evt_wait_or()可以使任务处于 WAIT_OR 状态
	WAIT_AND	等待最少所有事件标志的任务处于 WAIT_AND 状态。当这些标志被置位后，任务切换到就绪状态。函数 os_evt_wait_and()可以使任务处于 WAIT_AND 状态
	WAIT_SEM	等待信号量的任务处于 WAIT_SEM 状态。当获得信号量标志时，任务切换到就绪状态。函数 os_sem_wait()可以使任务处于 WAIT_SEM 状态
	WAIT_MUT	等待互斥量的任务处于 WAIT_MUT 状态。当对应的互斥量被释放时，任务切换到就绪状态。函数 os_mut_wait()可以使任务处于 WAIT_MUT 状态
	WAIT_MBX	等待接收信箱消息的任务处于 WAIT_MBX 状态。一旦获得消息，任务切换到就绪状态。函数 oos_mbx_wait()可以使任务处于 WAIT_MBX 状态。当信箱已满时，等待发送消息的任务也处于 WAIT_MBX 状态。当有任务从信箱中读出消息后，该发送任务切换到就绪状态。在这种情况下，函数 os_mbx_send()可以使任务处于 WAIT_MBX 状态
停用（INACTIVE）		没有开始运行的或已经被撤消了的任务处于 INACTIVE 状态。函数 os_tsk_delete()可以时通过 os_tsk_create()创造的任务处于 INACTIVE 状态

任务的创建是通过 os_tsk_create 函数创建的，os_tsk_creat 函数将创建由参数任务函数指针*task 指定的任务，并将任务添加就绪队列中，新任务会被动态分配一个任务识别号（TID）。下例中的 tsk1、tsk2 为任务的任务的 ID。task2 的任务在 task1 中通过 so_tsk_create 创建，并指定优先级为 1。task1 和 task2 通过__task 关键字声明。

```
#include <rtl.h>

OS_TID tsk1, tsk2;

__task    void task1 (void) {
   .
   .
    tsk2 = os_tsk_create (task2, 1);
   .
   .
}

__task void task2 (void) {
   for(;;)
   .
}
```

除了 os_tsk_create，另外还有 os_tsk_create_ex、os_tsk_create_user、os_tsk_create_user_ex 函数可用于创建新的任务，其中带有_user 后缀的是指由用户指定堆栈，带有_ex 后缀的是指传递参数给任务。任务的优先级除了可在创建时指定优先级，也可通过 os_tsk_prio 改变优先级。os_task_self()返回当前任务的任务 ID。

内核的启动代码如下：

```
#include <rt1.h>
void main () {
…
os_sys_init(task-name);
}
```

os_sys_init(task-name)负责初始化、启动 RTX 内核，创建优先级为 1 的 task-name 任务，运行 task-name 任务。另外一个函数 os_sys_init_prio(taskname, p)的功能与 os_sys_init 的功能相同，但指定优先级。

16.1.2　RTX 调度

RTX 的任务调度主要有协同多任务调度、抢占多任务调度、轮转多任务调度。

协同多任务调度是指任务调用自己的调度程序或者使自己进入休眠状态；操作系统不影响任务间的转换。

轮转多任务调度是指 RTX 核可被配置为运行并发任务情况。实际上，这些任务并非真正并发执行，而只是基于时间片的轮转，由于时间片很短，仅有几毫秒，所以这些任务给人感觉在同时执行。任务在自己的时间片里执行，当时间片用完后或任务放弃时间片后，系统就去执行下一个有同样优先级的就绪任务。此时，如果没有同样优先级的就绪任务，当前运行的任务会继续执行。时间片的长度可在RTX_config.c配置文件中设置。

抢占多任务处理是指调度程序可抢占当前运行的任务，调度程序决定下一个运行的任务。因此高优先级任务可抢占低优先级任务；被抢占任务需保存当前的现场并转成就绪态。发生以下情况时，当前任务被抢占。

- 任务调度器在系统时间片中断时执行。任务调度器处理任务的延迟：如果一个更高优先级任务的延迟超期了，更高优先级任务将取代当前执行的任务的运行。
- 通过当前执行的任务或中断服务程序将某个事件设置为更高优先级的任务，那么当前运行的任务会被挂起，这个更高优先级的任务将运行。
- 某个标记被返回给信号量，而又有一个更高优先级的任务等待该标记。这时当前运行的任务被挂起，等待该标记的任务开始执行。这个标记通过当前运行的任务或中断服务程序返回。
- 某互斥量被释放，且又有一个更高优先级的任务在等待该互斥量。这时当前运行的任务被挂起，等待该互斥量的任务开始执行。
- 某消息被传递到信箱，且又有一个更高优先级的任务在等待该消息。这时当前运行的任务被挂起，优先级高的任务开始执行。这个消息通过当前运行的任务或中断服务程序发送到信箱中。

- 信箱已满，且又有一个更高优先级的任务等待向信箱中发送消息。一旦当前的任务或中断服务程序从满信箱中取出一个消息，那么优先级高的任务就开始执行。
- 当前运行程序的优先级减小，如果又有其他任务就绪且优先级比当前运行的任务高，那么当前的任务会挂起，高优先级任务运行。

下面是一个多任务抢占的例子，任务 job1 比 job2 的优先级高。当 job1 开始执行时创建 job2，然后进入 os_evt_wait_or 函数；这时 job1 被挂起，job2 开始执行；一旦 job2 为 job1 设置事件标志后，job2 被挂起，job1 被唤醒执行；任务 job1 增加计数器 cnt1 的值，然后又调用 os_evt_wait_or()函数挂起自身，这时 job2 被唤醒，计数器 cnt2 加 1，同时为 job1 设置事件标志。通过串口可以观察到 cnt1 与 cnt2 交替出现，并自动加 1。

```
#include <rtl.h>
OS_TID tsk1,tsk2;
int    cnt1,cnt2;
__task void job1 (void);
__task void job2 (void);
__task void init(void)
{
    USART_Configuration();
    tsk1=os_tsk_create(job1,2) ;
    tsk2=os_tsk_create (job2, 1);
    os_tsk_delete_self();
}

__task void job1 (void) {
       while (1) {
         os_evt_wait_or (0x0001, 0xffff);
         cnt1++;
         printf("cnt1=%d\n\r",cnt1);

    }
}

__task void job2 (void) {
    while (1) {
        os_evt_set (0x0001, tsk1);
        cnt2++;
            printf("cnt2=%d\n\r",cnt2);
    }
}

void main (void) {
    os_sys_init (init);
    while (1);
}
```

16.2　任 务 通 信

while(1)的超级循环是通过全局变量方式进行功能模块的信息交互，RTX 是通过通信函数进行的。通信函数用于同步任务、管理公共资源（类似外设或者内存）、任务之间的消息传递。RTX 提供的任务通信方式有事件标志、信号量、互斥量、邮箱。

16.2.1　事件标志

事件标志是实现任务同步的主要方法，每个任务有 16 个事件标识可供使用，所以最多能等待 16 个不同的事件。也可以同时等待多个事件标志，这种情况下，如果这些事件标志是与的关系，那么这些事件标志必须都被置位后该任务才能继续运行；如果这些事件标志是或的关系，那么这些事件标志中的一个或几个被置位后该任务就可以继续运行。事件标志可被 ARM 中断功能置位。在这种机制下，通过使用 ARM 中断函数设置任务等待的标志，可以使异步的外部事件和 RTX 核的任务同步。

在 16.1.2 演示抢占调度已经使用事件标志实现了任务之间通信。将 16.1.2 节中的两个任务作以下修改。

```
__task void job1 (void) {
        while (1) {
           printf("begin    job1 \n\r");
           os_evt_wait_or (0x0001, 0xffff);
           printf ("end job1\n\r");

    }
}

__task void job2 (void) {
     while (1) {
           printf("begin job2\n\r");
                os_evt_set (0x0001, tsk1);
                printf("end job2\n\r");
     }
}
```

运行结果以及结果的分析如下。

begin　job1　　（1）job1 优先级高首先运行。
begin job2　　（2）job1 运行到 os_evt_wait_or，进入等待状态。运行 job2。
end job1　　（3）job2 运行 os_evt_set 后，job1 恢复。
begin　job1　　（4）job1 的 while 循环，回到起始，即（2）状态。
end job2　　（5）恢复（3）中 job2 的运行地方，继续执行。
begin job2　　（6）由于 job1 处于等待状态，使得 job2 继续执行，并执行 os_evt_set。
end job1　　（7）由于 job2 执行 os_evt_set 后，使得 job1 的继续执行。后续步骤同（4）。

除了程序中使用到的两个事件标志函数，另外有 4 个事件函数。表 16-3 给出了全部事件函数。

表 16-3　事件函数列表

事 件 函 数	描　　述
os_evt_clr	清除至少一个事件标志
os_evt_get	获取事件标志，使 os_evt_wait_or 运行
os_evt_set	设置至少一个事件标志
os_evt_wait_and	等待最少所有的事件标志被设置
os_evt_wait_or	等待至少一个事件标志被设置
isr_evt_set	从中断服务程序中设置至少一个事件标志

16.2.2　互斥量

互斥量用来控制临界区的访问，只有拥有互斥量的任务才能访问临界区，其他试图访问临界区的任务将被阻塞。图 16-2 是通过互斥量控制 USART 外设的例子。Task1 和 Task2 两个任务，拥有互斥量的任务才能调用 printf 函数。互斥量首先在 init 任务中通过 os_mut_init 初始化。而后在 Task1 与 Task2 中通过 os_mut_wait 与 os_mut_release 等待、释放互斥量。

需要注意的是程序中的 Task1 与 Task2 优先级相同。如果优先级不同，下面程序会造成优先级低的任务永远无法拿到互斥量。需要在每个任务中增加 os_dly_wait，以便让出 CPU 控制权。

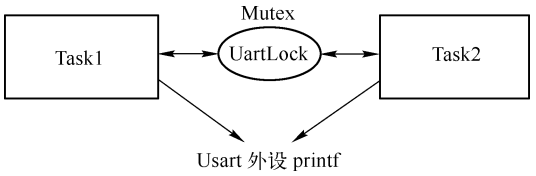


图 16-2　使用互斥控制资源共享

```
#include <rtl.h>
OS_TID tsk1,tsk2;
OS_MUT UartLock;
int    cnt1,cnt2;
__task void task1 (void);
__task void task2 (void);
__task void init(void)
{
os_mut_init(UartLock);
USART_Configuration();
tsk1=os_tsk_create(task1,1) ;
tsk2=os_tsk_create (task2, 1);
os_tsk_delete_self();
}
```

```
__task void task1 (void) {
while (1) {
    //os_dly_wait(50);
    os_mut_wait(UartLock, 0xFFFF); // 一直等待 UartLock 锁
    printf("task1\n\r" );
    os_mut_release(UartLock);      // 互斥释放

    }
}

__task void task2 (void) {
while (1) {
    //os_dly_wait(50);
    os_mut_wait(UartLock, 0xFFFF); // 一直等待 UartLock 锁
    printf("task2\n\r" );
    os_mut_release(UartLock);      // 互斥释放
    }
}

void main (void) {
os_sys_init (init);
while (1);
}
```

有关互斥的函数共三个，上例已全部使用。信号量类似互斥量。但互斥的局限是任务访问一个共享资源，信号量可限定一定数量的任务访问一系列共享资源。信号量可以实现多个同类资源的多任务互斥和同步。当信号量为单值信号量时，实现了互斥功能。表 16-4 是 RTX 的信号量函数，与互斥类似，但一个显著区别是 isr_sem_send 函数。

表 16-4　信号量函数

程　　序	描　　述
os_sem_init	初始化信号量对象
os_sem_send	发送一个信号（标志）给信号量
os_sem_wait	等待来自信号量的信号（标志）
isr_sem_send	发送一个信号（标志）给信号量

16.2.3　信箱

RTX 是通过邮箱在任务间传递复杂信息。首先在一个任务中定义数据集，而后将该数据集的指针传递给另外一个任务。RTX 中的邮箱支持多个消息，所以多个任务可同时发送消息到同一个邮箱中。定义一个邮箱时可确定邮箱中消息最大数目。邮箱中的消息是指包含协议消息或帧的内存块的指针，这样的内存块可以动态地分配和提供给用户。为了防止内存泄漏，用户需正确地分配和回收内存块。如果接收任务访问信箱中的消息不存在，它将被挂起，直到该消息被发送任务发送到信箱中，该被挂起的接收任务才会被唤醒。

RTX 的消息对象是指向一块内存的指针，对消息大小或内容无任何限制。RTX 内核处理消息的指针。

例如，定义一个整型变量的语句是 int message，则消息就是&message。当其他任务得到这个消息后（os_mbx_send (send_mbx, (void *)&message, 0xffff)），就得到了 message 变量的指针，可以对 message 变量进行读和写操作。

为了发送固定大小的消息，需要从动态内存分配内存块，发送内存块的指针给邮箱。接收任务收到指针后，读取内存块信息，并释放内存块。RTX 核有一个功能强大的固定内存块内存分配函数。它们是线程安全、可重入，且能被 RTX 核无限制地使用。建议使用固定内存块分配函数发送固定大小的消息。需要通过_init_box 函数为消息对象初始化内存池。

下面的例子是首先在 send_task 中创建消息，而后通过邮箱发送给 rec_task。

```
#include <rtl.h>
os_mbx_declare (MsgBox, 16);                    /* 定义 RTX 邮箱*/
_declare_box(mpool,12,8);                       /* 12 块，每块 8 个字节 */
__task void rec_task (void);
//发送消息
__task void send_task (void) {
  U32 *mptr;
  os_tsk_create (rec_task, 0);
  os_mbx_init (MsgBox, sizeof(MsgBox));
  mptr = _alloc_box (mpool);                    /* 给消息分配一个内存 */
  mptr[0] = 0x12345678;                         /* 设置消息内容 */
  mptr[1] = 0x87654321;
  os_mbx_send (MsgBox, mptr, 0xffff);           /* 发送消息到邮箱 */
  os_tsk_delete_self ();
}
/*接收消息*/
__task void rec_task (void) {

  U32 *rptr, rec_val[2];

  os_mbx_wait (MsgBox, (void **) &rptr, 0xffff);    /* 等待邮箱 */
  rec_val[0] = rptr[0];                             /* 保存消息中的数值 */
  rec_val[1] = rptr[1];
printf ("0x%X   ;0x%X\n\r",rec_val[0],rec_val[1]);
  _free_box (mpool, rptr);                          /* 释放内存块*/
  os_tsk_delete_self ();
}

void main (void) {
USART_Configuration();
  _init_box (mpool, sizeof(mpool), sizeof(U32));    /*初始化固定块大小的存储池*/
  os_sys_init(send_task);
}
```

表 16-5 是关于邮箱与内存管理程序的函数以及说明。

表 16-5　邮箱与内存管理函数

函　　数	描　　述
os_mbx_check	可再加入信箱的消息数
os_mbx_declare	创建信箱对象
os_mbx_init	初始化信箱
os_mbx_send	给信箱发送消息
os_mbx_wait	等待邮箱中的消息，如果消息可用，读取消息指针
isr_mbx_receive	从信箱中取出下一条消息
isr_mbx_send	给信箱发送消息
_declare_box	采用 4 位对齐方式创建固定大小块的内存池
_declare_box8	采用 8 位对齐方式创建固定大小块的内存池
_init_box	初始化 4 位对齐的内存池
_init_box8	初始化 8 位对齐的内存池
_alloc_box	从内存池中分配一个内存块
_calloc_box	从内存池中分配一个内存块，并初始化其值为 0
_free_box	回收内存块

16.3　RTX 基础配置

要使用 RTX，首先需要在工程文件中加载 RTX 内核，如图 16-3 所示。

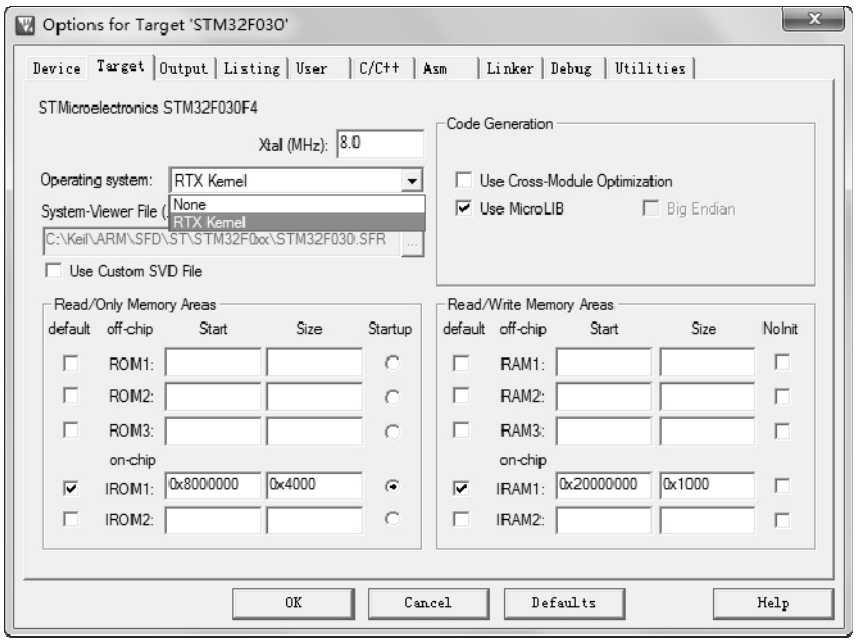


图 16-3　RTX 内核配置

在 RTX 的应用程序中，需要包含 MDK 提供的“RTL.h”文件与 RTX_Config.c 文件。在\Keil\ARM\RL\RTX\Config 目录下提供了多个 RTX_Config.c 的模板，针对 ARM7™/ARM9™有多个版本的 RTX_config.c 文件。但对于 Cortex™-M 只有一个 RTX_Conf_CM.c 文件，该文件对所有的 Cortex™-M 是相同的。RTX_config.c 主要包含当前运行任务的数目配置、任务堆栈配置、CPU 的时钟、操作系统的节拍时钟，以及是否支持轮转任务调度、用户定时器设置信息。表 16-6 是 RTX_Conf_CM.c 文件中的定义解释以及本章中使用数值。

表 16-6　RTX_Conf_CM.c 配置

宏 定 义	数　值	描　述
OS_TASKCNT	6	当前运行的任务数，包括了除不活动状态外所有任务（运行、等待、就绪）
OS_PRIVCNT	0	用户提供栈的任务数量
OS_STKSIZE	63	任务堆栈大小
OS_STKCHECK	1	是否检查堆栈溢出
OS_RUNPRIV	1	运行特权模式
OS_TIMER	0	时钟定时器配置，选用内核 SysTick 或者外设定时器
OS_CLOCK	48000000	定时器的输入时钟频率
OS_TICK	10000	周期 RTX 中断的时间间隔，默认 10000 值为 10ms
OS_ROBIN	1	使能轮转调度
OS_ROBINTOUT	5	轮转超时间隔
OS_TIMERCNT	0	用户定时器数量
OS_FIFOSZ	16	ISR FIFO 队列容量

注意：如果是基于 STM32F0 的固件库设计 RTX 程序，需要将 stm32f0xx_it.c 中的 SVC_Handler 与 PendSV_Handler 函数内容删除。因为在 RTX 内核中已经定义该部分内容。

16.4　中断任务之间的通信实例

实时操作系统与其他操作系统的重要区别就是需满足处理与时间的关系，即在实时计算时，系统的正确性不仅仅依赖于计算的逻辑结果而且依赖于结果产生的时间。要求实时操作系统必须在规定的时间内完成对外部或者内部事件的响应，即实时操作系统对时间要求是确定性的。然而中断属于硬件机制，用于通知 CPU 异步事件发生，中断是不确定的。所以良好的中断服务程序应尽可能地短小。对于需要由中断触发的复杂处理，可通过 isr_为前缀的事件、信号量和邮箱等通知较高优先级的任务处理中断中的未完成的工作。

由于 printf、malloc 的函数可能被阻塞，中断服务程序中不能包含可能被阻塞的函数。16.4 节中通过 printf 函数显示任务运行状况的方式在中断程序是不可行的。基于第 7 章的 USART 程序的实例程序修改，主要程序如下。

```
        /*模拟普通任务运行过程中出现中断。为了保证中断发生在该任务执行过程中，使用了软件产生中断方式*/
        __task void InsertInterrupt (void) {
        for (;;)
```

```
    {
            UART_Send("before\n\r",8);
            EXTI_GenerateSWInterrupt(EXTI_Line0);
            UART_Send("after\n\r",7);
    }
}
/*等待中断发来的事件*/
__task void InterruptTask(void) {
int i =0;

 OS_RESULT result;
 for (;;)
{
            result =os_evt_wait_or (0x0001, 0xffff);     /*  等待时间标志 0x0001         */
          if (result == OS_R_TMO) {//超时，未等待到
      ;
    }
    else {
      UART_Send("insert\n\r",8);
    }
  }
}
/*  初始化任务  */
__task void init (void) {

t_Task= os_tsk_create (InterruptTask, 0);   /*  与中断关联的任务优先级比较高    */
  t_NormalTask = os_tsk_create (InsertInterrupt, 0);   /*  普通任务*/
  EXTI0_Config(); //配置 PA0 中断
os_tsk_delete_self ();
}
```

在中断服务程序中，有关事件标志、邮箱、信号量的函数是以 isr 为前缀的函数，而不是 os 开头的函数。中断函数如下。

```
void EXTI0_1_IRQHandler(void)
{
  if(EXTI_GetITStatus(EXTI_Line0) != RESET)
  {
    UART_Send("In Interrupt\n\r",14);
    isr_evt_set (0x0001, t_Task);
    EXTI_ClearITPendingBit(EXTI_Line0);
  }
}
```

下面部分显示运行的结果（由于数据显示较快。该程序的 PC 端，建议使用 PuTTY。因为部分网上的串口软件显示接收数据慢半拍），即在 InsertInterrupt 任务过程被中断挂起，中断运行结束后，由于 InterruptTask 任务优先级高，故执行就绪状态的 InterruptTask 任务，而

不是回到原来的 InsertInterrupt 任务。在 InterruptTask 执行完毕后，才回到原来的 InsertInterrupt 任务。图 16-4 演示了该过程。

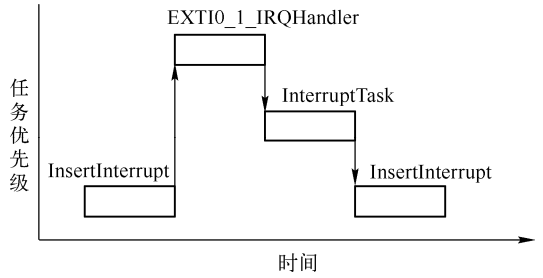


图 16-4　任务、中断之间的抢占

```
before
In Interrupt
insert
after
```

针对这个运行结果不妨将两个任务的优先级改成一样，观察一下运行结果。

16.5　小　　结

本章首先探讨了实时操作系统与大循环模式的区别；而后分别介绍了 RTX 内核的任务创建以及任务之间通信，并给出相应例子；最后以实例方式演示了因中断触发任务抢占。

第17章

USB 电源监测

由于 USB 接口不仅带有高速数据通信功能还对外提供 5V 电源，所以可用于各种通信设备（如手机）的充电。本章将围绕如何利用 STM32F0x 来实现对 USB 接口电压/电流的采集、监测。为此构建了满足 STM32F0x 的 ADC 端口电压要求的电路。针对对电流和电压两路信号同时采样的问题，讲解了选用 DMA 而非中断的原因。

17.1 需 求 分 析

USB 3.0 标准要求接口供电能力为 1A，USB 2.0 标准要求接口供电能力为为 0.5A。USB 的输出电压均为 5V。本章主要以 USB 2.0 标准要求的 0.5A 电流为基础分析如何检测电流和电压。

17.2 硬 件 设 计

图 17-1 是 USB 电压、电流采样电路图。USB 的电源正常电压为 5V，经分压后，进入 STM32F0 PA0.2 的电压为 2.5V，未超过 STM32F0 的 ADC 端口电压 3.3V 限制。

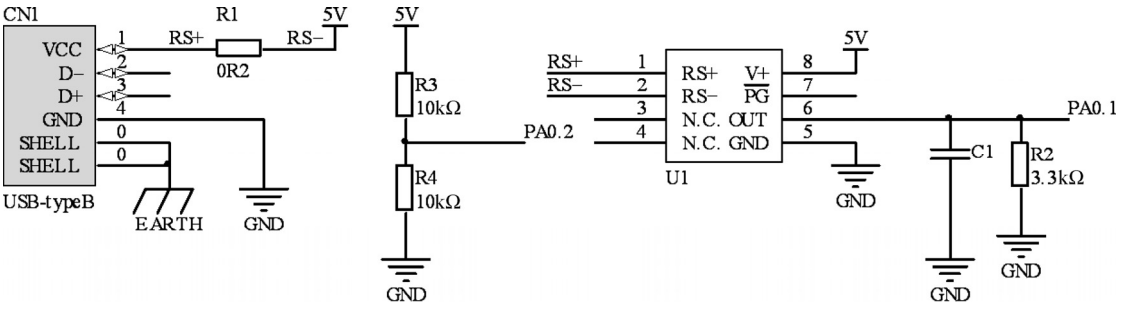


图 17-1　USB 电压电流采样图

电流测量通过电流差采样电阻 R 与电流放大器 U1（MAX4172）获得。采样电阻 R1 为 0.2Ω。MAX4172 是电流检测放大器，是通过外接测流电阻 R1 检测负载电流，改变测流电阻

值（R1）可实现对不同负载电流检测，U1 输出为电流，可由一个接地电阻 R2 转化为对地电压，通过改变接地电阻 R2 的阻值提供大范围的输出电压和电流。

R2 两端电压= $G_M \times R_1 \times R_2 \times I_{R1}$，其中 G_M 为互导系数，MAX4172 为 10mA/V。表 17-1 为 USB 电流换算得到的 R2 两端的电压值以及对应 12 位 ADC 采样值。

表 17-1　USB 电流与采样电压对应关系

USB 电流	电流采样电阻 R1 两端电压	放大器输出电流放大器输出电压	放大器输出电压值（R2）	ADC 读数值
0mA	0mV	0mA	0V	0x000
250mA	50mV	0.5mA	1.65V	0x800
500mA	100mV	1mA	3.3V	0xFFF

17.3　软 件 设 计

程序采用通过 TIM1 启动 ADC 的方式，如果使用 ADC 采样中断方式读取两路 ADC 通道采样结果，一次采集需要连续两次中断服务程序。所以采用与 11 章的实例不同的读取采样结果方式——DMA 方式，即结果由 DMA 读取并产生 DMA 中断。在 DMA 中断将采集结果赋值，如果需要计算有功、总功、有效值计算可在 DMA 中断计算或者通知超级大循环（while(1)）进行累加和计算。TIM_Config()被用来配置 ADC 采样时间间隔。ADC_Config()用于配置 ADC 采样通道，需注意的是 ADC_InitStructure.ADC_ContinuousConvMode 与 ADC_InitStructure.ADC_ExternalTrigConv 参数配置，ADC_DMACmd(ADC1, ENABLE)启动了 ADC 的 DMA 方式。DMA_Config()设置了 DMA 通道。具体代码如下：

```
static void TIM_Config(void)
{
  TIM_TimeBaseInitTypeDef   TIM_TimeBaseStructure;
  TIM_OCInitTypeDef         TIM_OCInitStructure;

  /* TIM1 外设时钟使能 */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1 , ENABLE);

  TIM_DeInit(TIM1);
  TIM_TimeBaseStructInit(&TIM_TimeBaseStructure);
  TIM_OCStructInit(&TIM_OCInitStructure);

  /* 时基配置 */
  TIM_TimeBaseStructure.TIM_Period = 0xFFF1;
  TIM_TimeBaseStructure.TIM_Prescaler = 0xFF ;
  TIM_TimeBaseStructure.TIM_ClockDivision = 0x0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);

  /* 输出比较 PWM 配置 */
```

```
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; /* 默认低电平 */
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_Pulse = 0x01;
    TIM_OC4Init(TIM1, &TIM_OCInitStructure);

    /* TIM1 使能计数器*/
    TIM_Cmd(TIM1, ENABLE);

    /*输出使能*/
    TIM_CtrlPWMOutputs(TIM1, ENABLE);
}

static void ADC_Config(void)
{
    ADC_InitTypeDef       ADC_InitStructure;
    GPIO_InitTypeDef      GPIO_InitStructure;
    ADC_DeInit(ADC1);
    /* GPIOA 外设时钟使能 */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
     /* ADC1 时钟使能 */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1|GPIO_Pin_2 ;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    /* 初始化 ADC*/
    ADC_StructInit(&ADC_InitStructure);

      /* 配置 ADC1 分辨率为 12 位 */
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE ;//每次转换均有时钟触发，不采用连
续转换方式
    ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_Rising;//外部触发
    ADC_InitStructure.ADC_ExternalTrigConv =   ADC_ExternalTrigConv_T1_CC4;//T1 触发
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;//12 位数据右对齐
    ADC_InitStructure.ADC_ScanDirection = ADC_ScanDirection_Upward;

    ADC_Init(ADC1, &ADC_InitStructure);
    ADC_ChannelConfig(ADC1, ADC_Channel_1 , ADC_SampleTime_55_5Cycles);
ADC_ChannelConfig(ADC1, ADC_Channel_2 , ADC_SampleTime_55_5Cycles);
    /* 校准 */
    ADC_GetCalibrationFactor(ADC1);
    /* ADC DMA 循环方式的 DMA */
    ADC_DMARequestModeConfig(ADC1, ADC_DMAMode_Circular);
    /*使能 ADC_DMA */
    ADC_DMACmd(ADC1, ENABLE);
```

```
    /*使能 ADC 外设 */
    ADC_Cmd(ADC1, ENABLE);
    /* 等待 ADRDY 标志 */
    while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_ADRDY));
    /* ADC1 regular Software Start Conv */
    ADC_StartOfConversion(ADC1);
}

/* DMA 通道配置 */
static void DMA_Config(void)
{
    DMA_InitTypeDef     DMA_InitStructure;
    /* DMA1 时钟设置 */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1 , ENABLE);
    /* DMA1 通道配置 */
    DMA_DeInit(DMA1_Channel1);
    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)ADC1_DR_Address;//外设地址
    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t) RegularConvData_Tab;
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
    DMA_InitStructure.DMA_BufferSize = 2;
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
    DMA_Init(DMA1_Channel1, &DMA_InitStructure);
// 设置 DMA 中断
DMA_ITConfig(DMA1_Channel1,DMA_IT_TC,ENABLE );
    NVIC_InitStructure.NVIC_IRQChannel = DMA1_Channel1_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPriority = 0x01;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
//
    /* DMA1 通道使能 */
    DMA_Cmd(DMA1_Channel1, ENABLE);
    }
```

在 DMA 中断服务程序中，将采样结果读取，可在此进行累加和计算用于电流/电压的有效值、总功以及通过 vol 与 cur 的乘积累加和计算有功。中断服务程序如下：

```
void DMA1_Channel1_IRQHandler(void)
{
 if (DMA_GetITStatus(DMA1_IT_TC1)!= RESET)
 { endflag=1;
     v=RegularConvData_Tab[0] * 500/ 0xFFF; //电流，单位为 mA
```

```
                cur=RegularConvData_Tab[1] * 3300*2 / 0xFFF   ; //电压，单位为 mV
                DMA_ClearITPendingBit(DMA1_IT_TC1);
            }
        }
```

17.4　小　　结

本章通过对 USB 的电流和电压检测的实例，介绍了如何根据需求以及 STM32F0x 的 ADC 特点要求，进行电路设计；并根据 STM32F0x 的 ADC 采样通道特点选用符合需求的 DMA 读取 ADC 采样结果方式，说明了功率的计算。

反侵权盗版声明